Broadview



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Effective C++ 中文版

Third Edition 第三版

55 Specific Ways to Improve Your Programs and Designs

改善程序与设计的55个具体做法



[美] Scott Meyers 著

侯捷 译



電子工業出版社

目錄

Effective C++

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

来源：http://blog.csdn.net/fatalerror99/

Preface（前言）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我在 1991 年写出了 *Effective C++* 的最早版本，1997 年出了第二版，我更新了一些重要的方面的素材，但是，因为我不想使熟悉本书第一版的读者感到困惑，我尽最大可能保持了原有的结构。最早的 50 个 Item 标题中的 48 个原则上保持不变。如果把书看作一栋房子，第二版就相当于通过更换地毯、涂料和灯光设备等使其焕然一新。

对于第三版，我打散了原有的秩序。（很多次我希望我能做得更彻底。）1991 年以来，C++ 世界经历了巨大的改变，而将近 15 年以前我制定的那些 Item 也不再契合本书的目标——将最重要的 C++ 编程准则融入小而易读的建议中。1991 年，假设 C++ 程序员具有 C 语言背景是有一定道理的。现在，转到 C++ 的程序员很可能来自 Java 或 C#。1991 年，inheritance（继承）和 object-oriented programming（面向对象编程）对于大多数程序员来说都是新鲜的。现在，这些已经是非常普遍的概念，而 exceptions（异常），templates（模板），以及 generic programming（泛型编程）成为新的需要更多指导的领域。1991 年，没有人听说过 design patterns（设计模式）。现在，讨论软件系统时很难不涉及它们。1991 年，C++ 正式标准化的工作刚刚开始。现在，标准化已经八年了，而下一个版本的工作也已经开始。

为应对这些变化，我尽己所能将写字板擦得一干二净，并不断地追问自己："在 2005 年，对于目前的 C++ 程序员，什么才是最重要的建议？"结果就是这个新版本中的一组 Item。本书包括了关于 resource management（资源管理）和 programming with templates（使用模板编程）的新的章节。实际上，templates（模板）的考虑贯穿全书，因为它几乎影响了 C++ 的每个方面。本书也包括关于在 exceptions（异常）存在的场合下编程，在应用 design patterns（设计模式），以及在使用新的 TR1 库程序（TR1 在 Item 54 中介绍）等方面的新的素材。本书还承认在 single-threaded systems（单线程系统）中能很好地工作的技术和方法可能不适用于 multithreaded systems（多线程系统）。噢，本书超过一半的素材都是新的。然而，第二版中基本原理方面的资料中的大部分依然是重要的，所以我将它们以这样或那样的形式保留下来。（你能在 Appendix B（附录 B）找到第二版和第三版 Item 的对照表。）

我尽我所能使本书趋于最好，但我并不幻想完美。如果你觉得本书中的一些 Item 作为常规的建议不太合适；或者有更好的方法实现本书中需要完成的任务；或者有一个或更多技术上的讨论不明确，不完全，或容易令人误解，请告诉我。如果你发现任何错误——技术上的，文法上的，印刷上的，无论哪种——也请告诉我。如果你是第一个让我注意到某个问题的人，我很高兴将你的名字加入到以后再次印刷的致谢中。

即使 Item 的数量增加到 55，本书中的这一组准则离完满无遗还差得很远。但是成为好的规则——在几乎所有的应用程序几乎所有的时间中得到应用——比它看上去更加困难。如果你有增加准则的建议，我很高兴能听到它。

我维护本书从第一次印刷起的变化列表，包括错误修正，进一步解释，和技术更新。这个列表可以从 *Effective C++ Errata* 网页得到，http://aristeia.com/BookErrata/ec++3e-errata.html。如果你希望当我更新列表时，你能得到通报，我建议你参加我的 mailing list（邮件列表）。我用它作为通告，发给那些对跟踪我的职业工作感兴趣的人们。关于细节问题，请参考 http://aristeia.com/MailingList/。

SCOTT DOUGLAS MEYERS STAFFORD, OREGON http://aristeia.com/ APRIL 2005

Introduction（导言）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

学习一种编程语言的基础是一回事；学习如何用那种语言设计和实现高效率的程序完全是另外一回事。对于 C++ ——一种以拥有非同寻常的能力范围和表现力而自豪的语言——更是尤其如此。如果能正确使用，与 C++ 共事是一件令人快乐的事情。极多样的设计样式被直接表达并有效实现。对于 classes（类），functions（函数），以及 templates（模板）的明智选择和小心精巧的安排能使应用程序的编程更加简单，直观，高效，并基本无错。如果你知道如何去做，写出高效的 C++ 程序并不特别难。然而，如果不经训练就贸然使用，C++ 也会导致不可理解的，难以维护的，无法扩展的，低效率的，错误百出的代码。

本书的目的在于引导你如何高效使用 C++。我假设你已经熟悉了作为语言的 C++ 并有使用它的一些经验。我在此提供的是使用这种语言的指南，以使你的程序易于理解，可维护，易移植，可扩展，效率高，而且行为符合你的预期。

我提供的建议落在两个主要的范围中：普通的设计策略，以及特殊语言特性的具体细节。设计的讨论集中于如何在 C++ 做某件事情的多种不同方法之间进行选择。如何在 inheritance（继承）和 templates（模板）之间选择？如何在 public（公有）和 private inheritance（私有继承）之间选择？如何在 private inheritance（私有继承）和 composition（复合）之间选择？如何在 member（成员）和 non-member functions（非成员函数）之间选择？如何在 pass-by-value（传值）和 pass-by-reference（传引用）之间选择？在一开始就做出正确的决定是很重要的，因为一个不好的选择可能会直到开发过程很晚的阶段才显现出来，在这时候再调整它常常是困难重重，极为耗时而且代价不菲的。

即使在你正确地知道你要做什么的时候，仅仅把事情做对也是需要技巧的。assignment operators（赋值运算符）的合适的返回类型是什么？destructor（析构函数）什么时候应该是 virtual（虚拟）的？当 operator new（运算符 new）找不到足够的内存时它应该怎么办？类似这些的令人费神的细节是至关重要的，因为错误的做法几乎总是导致无法预料的，很可能令人迷惑的程序行为。这本书正是来帮助你避免这些问题的。

这不是一本全面的 C++ 手册。它收集了 55 个详细的提议（我将它们称为 Items）告诉你怎样才能改善你的程序和设计。每一个 Item 都能 stands more or less on its own（独立成章），但大部分也包含对其它 Items 的参考。因而，读这本书的一个方法就是从你感兴趣的一个 Item 开始，然后顺着它的参考条目继续看下去。

这本书也不是一本 C++ 入门书。例如，在 Chapter 2（第二章），我希望能告诉你关于 constructors（构造函数），destructors（析构函数），以及 assignment operators（赋值运算符）正确实现的全部内容，但是我假设你已经知道或者能在别处找到这些函数做些什么以及它们该怎样声明的资料。大量 C++ 书籍包含类似这样的信息。

这本书的目的是为了突出 C++ 编程中那些常常被忽略的方面。其它书描述了语言的各个部分。这本书则告诉你如何将这些部分结合起来以达到高效编程的最终目的。其它书告诉你如何使你的程序能够编译。这本书则告诉你如何避免那些编译器不能告诉你的问题。

与此同时，本书将自己限制在 standard C++（标准 C++）之内。在此仅仅使用官方的语言标准中的特性。可移植性是本书一个关键的关注点，所以如果你要寻找平台依赖的特性和部件，在这里不会找到。

另一个在本书中找不到的东西是 C++ Gospel（C++ 福音书）——通向完美的 C++ 软件的一条 One True Path（真理之路）。本书中的每一个 Item 都在如何生成更好的设计，如何避免一般的问题，以及如何得到更高的效率等方面提供指导，但没有一个 Item 是普遍适用的。软件设计和实现是一项复杂的任务，被 hardware（硬件），operating system（操作系统），和 application（应用程序）的限制所影响，所以我能做的最好的事情就是为创建更好的程序提供 guidelines（指导方针）。

如果你在所有的时候都遵守这些 guidelines（指导方针），你将不太可能落入环绕在 C++ 周围的大量陷阱中，但是作为 guidelines（指导方针）的固有限制，它总有例外。这就是为什么每一个 Item 都有一个详细的解释。这些解释是本书中最重要的部分。只有理解了一个 Item 背后的基本原理，你才能决定它是否适合你开发的软件以及你被困其下的特有的限制。

本书最好用来增加关于 C++ 如何运转，为什么它会这样运转，以及如何让它的行为为你服务等方面的知识。盲目运用本书的 Items 无疑是不适当的，但是与此同时，如果你没有更好的理由，或许也不应该违反这些 guidelines（指导方针）中的任何一条。

Terminology（术语）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

这是一个所有程序员都应该了解的小型的 C++ vocabulary（词汇表）。下面的术语都足够重要，对它们的含义取得完全一致对于我们来说是完全必要的。

declaration（声明）告诉编译器关于某物的 name（名字）和 type（类型），但它省略了某些细节。以下这些都是 declaration（声明）：

extern int x; // object declaration

std::size_t numDigits(int number); // function declaration

class Widget; // class declaration

template<typename T> // template declaration

class GraphNode; // (see Item 42 for info on

// the use of "typename")

注意即使是 built-in type（内建类型），我还是更喜欢将整数 x 看作一个 "object"，某些人将 "object" 这个名字保留给 user-defined type（用户定义类型），但我不是他们中的一员。再有就是注意函数 numDigits 的返回类型是 std::size_t，也就是说，namespace（命名空间）std 中的 size_t 类型。这个 namespace（命名空间）是 C++ 标准库中每一样东西实际所在的地方。但是，因为 C 标准库（严谨地说，来自于 C89）在 C++ 中也能使用，从 C 继承来的符号（诸如 size_t）可能存在于全局范围，或 std 内部，或两者都有，这依赖于哪一个头文件被 #include。在本书中，我假设 C++ 头文件被 #include，这也就是为什么我用 std::size_t 代替 size_t 的原因。当文字讨论中涉及到标准库组件时，我一般不再提及 std，这依赖于你认可类似 size_t，vector，以及 cout 之类的东西都在 std 中，在示例代码中，我总是包含 std，因为真正的代码没有它将无法编译。

顺便说一下，size_t 仅仅是供 C++ 对某物计数时使用的某些 unsigned 类型的 typedef（例如，一个 char*-based string（基于 char* 的 string）中字符的个数，一个 STL container（容器）中元素的个数，等等）。它也是 vector，deque，和 string 的 operator[] 函数所持有的类型，这是一个在 Item 3 中定义我们自己的 operator[] 函数时将要遵守的惯例。

每一个函数的 declaration（声明）都表明了它的 signature（识别特征），也就是它的参数和返回类型。一个函数的 signature（识别特征）与它的类型相同。对于 numDigits 的情况，signature（识别特征）是 std::size_t (int)，也就是说，“函数取得一个 int，并返回一个

std::size_t”。官方的 "signature"（识别特征）的 C++ 定义排除了函数的返回类型，但是在本书中，将返回类型考虑为 signature（识别特征）的一部分更加有用。

definition（定义）为编译器提供在 declaration（声明）时被省略的细节。对于一个 object（对象），definition（定义）是编译器为 object（对象）留出内存的地方。对于一个 function（函数）或一个 function template（函数模板），definition（定义）提供代码本体。对于一个 class（类）或一个 class template（类模板），definition（定义）列出了 class（类）或者 template（模板）的 members（成员）：

```
int x; // object definition

std::size_t numDigits(int number) // function definition.

{ // (This function returns

std::size_t digitsSoFar = 1; // the number of digits

// in its parameter.)

while ((number /= 10) != 0) ++digitsSoFar;

return digitsSoFar;

}

class Widget { // class definition

public:

Widget();

~Widget();

...

};

template<typename T> // template definition

class GraphNode {

public:

GraphNode();

~GraphNode();

...

};
```

initialization（初始化）是设定一个 object（对象）的第一个值的过程。对于 user-defined types（用户定义类型）的 objects（对象），initialization（初始化）通过 constructors（构造函数）完成。default constructor（缺省构造函数）就是不需要任何 arguments（引数）就可以调用的那一个。这样的一个 constructor（构造函数）既可以是没有 parameters（参数），也可以是每一个 parameter（参数）都有缺省值：

```
class A {

public:

A(); // default constructor

};

class B {

public:

explicit B(int x = 0, bool b = true); // default constructor; see below

}; // for info on "explicit"

class C {

public:

explicit C(int x); // not a default constructor

};
```

这里 classes B 和 C 的 constructors（构造函数）都被声明为 explicit（显式）。这是为了防止它们被用来执行 implicit type conversions（隐式类型转换），虽然他们还可以被用于 explicit type conversions（显示类型转换）：

```
void doSomething(B bObject); // a function taking an object of
// type B
B bObj1; // an object of type B
doSomething(bObj1); // fine, passes a B to doSomething
B bObj2(28); // fine, creates a B from the int 28
// (the bool defaults to true)
doSomething(28); // error! doSomething takes a B,
// not an int, and there is no
// implicit conversion from int to B
doSomething(B(28)); // fine, uses the B constructor to
// explicitly convert (i.e., cast) the
// int to a B for this call. (See
// Item 27 for info on casting.)
```

constructors（构造函数）被声明为 explicit（显式）通常比 non-explicit（非显式）更可取，因为它们可以防止编译器执行意外的（常常是无意识的）type conversions（类型转换）。除非我有一个好的理由允许一个 constructor（构造函数）被用于 implicit type conversions（隐式类型转换），否则我就将它声明为 explicit（显式）。我希望你能遵循同样的方针。

请注意我是如何突出上面的示例代码中的 cast（强制转换）的。贯穿本书，我用这样的突出引导你注意那些应该注意的材料。（我也突出章节号码，但那仅仅是因为我想让它好看一些。）

copy constructor（拷贝构造函数）被用来以一个 object（对象）来初始化同类型的另一个 object（对象），copy assignment operator（拷贝赋值运算符）被用来将一个 object（对象）中的值拷贝到同类型的另一个 object（对象）中：

```
class Widget {
public:
Widget(); // default constructor
Widget(const Widget& rhs); // copy constructor
Widget& operator=(const Widget& rhs); // copy assignment operator
...
};
Widget w1; // invoke default constructor
Widget w2(w1); // invoke copy constructor
w1 = w2; // invoke copy
// assignment operator
```

当你看到什么东西看起来像一个 assignment（赋值）的话，要仔细阅读，因为 "=" 在语法上还可以被用来调用 copy constructor（拷贝构造函数）：

```
Widget w3 = w2; // invoke copy constructor!
```

幸运的是，copy constructor（拷贝构造函数）很容易从 copy assignment（拷贝赋值）中区别出来。如果一个新的 object（对象）被定义（就象上面那行代码中的 w3），一个 constructor（构造函数）必须被调用；它不可能是一个 assignment（赋值）。如果没有新的 object（对象）被定义（就象上面那行 "w1 = w2" 代码中），没有 constructor（构造函数）能被调用，所以它就是一个 assignment（赋值）。

copy constructor（拷贝构造函数）是一个特别重要的函数，因为它定义一个 object（对象）如何 passed by value（通过传值的方式被传递）。例如，考虑这个：

```
bool hasAcceptableQuality(Widget w);

...

Widget aWidget;

if (hasAcceptableQuality(aWidget)) ...
```

参数 w 通过传值的方式被传递给 hasAcceptableQuality，所以在上面的调用中，aWidget 被拷贝给 w。拷贝动作通过 Widget 的 copy constructor（拷贝构造函数）被执行。pass-by-value（通过传值方式传递）意味着 "call the copy constructor"（调用拷贝构造函数）。（然而，通过传值方式传递 user-defined types（用户定义类型）通常是一个不好的想法，pass-by-reference-to-const（传引用给 const）通常是更好的选择。关于细节，参见 Item 20。）

STL 是 Standard Template Library（标准模板库），作为 C++ 标准库的一部分，致力于 containers（容器）（例如，vector，list，set，map，等等），iterators（迭代器）（例如，vector<int>::iterator，set<string>::iterator，等等），algorithms（算法）（例如，for_each，find，sort，等等），以及相关机能。相关机能中的很多都通过 function objects（函数对象）——行为表现类似于函数的 objects（对象）——提供。这样的 objects（对象）来自于重载了 operator() ——函数调用运算符——的 class（类），如果你不熟悉 STL，在读本书的时候，你应该有一本像样的参考手册备查，因为对于我来说 STL 太有用了，以至于不能不利用它。一但你用了一点点，你也会有同样的感觉。

从 Java 或 C# 那样的语言来到 C++ 的程序员可能会对 undefined behavior（未定义行为）的概念感到吃惊。因为各种各样的原因，C++ 中的一些 constructs（结构成分）的行为没有确切的定义：你不能可靠地预知运行时会发生什么。这里是两个带有 undefined behavior（未定义行为）的代码的例子：

```
int *p = 0; // p is a null pointer

std::cout &lt;&lt; *p; // dereferencing a null pointer

// yields undefined behavior

char name[] = "Darla"; // name is an array of size 6 (don't

// forget the trailing null!)

char c = name[10]; // referring to an invalid array index

// yields undefined behavior
```

为了强调 undefined behavior（未定义行为）的结果是不可预言而且可能是令人讨厌的，有经验的 C++ 程序员常常说带有 undefined behavior（未定义行为）的程序 can（能）毁掉你的辛苦工作的成果。这是真的：一个带有 undefined behavior（未定义行为）的程序 could（可以）毁掉你的心血。只不过可能性不太大。更可能的是那个程序的表现反复无常，有时会运行正常，有时会彻底完蛋，还有时会产生错误的结果。有实力的 C++ 程序员能以最佳状态避开 undefined behavior（未定义行为）。本书中，我会指出许多你必须要注意它的地方。

另一个可能把从其它语言转到 C++ 的程序员搞糊涂的术语是 interface（接口）。Java 和 .NET 的语言都将 Interfaces（接口）作为一种语言要素，但是在 C++ 中没有这种事，但是在 Item 31 讨论了如何模拟它。当我使用术语 "interface"（接口）时，一般情况下我说的是一个函数的 signature（识别特征），是一个 class（类）的可访问元素（例如，一个 class（类）的 "public interface"，"protected interface"，或 "private interface"），或者是对一个 template（模板）的 type parameter（类型参数）来说必须合法的 expressions（表达式）（参见 Item 41）。也就是说，我是作为一个相当普遍的设计概念来谈论 interface（接口）的。

client（客户）是使用你写的代码（一般是 interfaces（接口））的某人或某物。例如，一个函数的 clients（客户）就是它的使用者：调用这个函数（或持有它的地址）的代码的片段以及写出和维护这样的代码的人。class（类）或者 template（模板）的 clients（客户）是使用这个 class（类）或 template（模板）的软件的部件，以及写出和维护那些代码的程序员。在讨论 clients（客户）的时候，我一般指向程序员，因为程序员会被困扰和误导，或者因为不好的 interfaces（接口）而烦恼。但他们写的代码却不会。

你也许不习惯于为 clients（客户）着想，但是我会用大量的时间试图说服你：你应该尽你所能使他们的生活更轻松。记住，你也是一个其他人开发的软件的 client（客户）。难道你不希望那些人为你把事情弄得轻松些吗？除此之外，你几乎肯定会在某个时候发现你自己处在了你自己的 client（客户）的位置上（也就是说，使用你写的代码），而这个时候，你会为你在开发你的 interfaces（接口）时在头脑中保持了对 client（客户）的关心而感到高兴。

在本书中，我常常掩盖 functions（函数）和 function templates（函数模板）之间以及 classes（类）和 class templates（类模板）之间的区别。那是因为对其中一个确定的事对另一个常常也可以确定。如果不是这样，我会区别对待 classes（类），functions（函数），以及由 classes（类）和 functions（函数）产生的 templates（模板）。

在代码注释中提到 constructor（构造函数）和 destructors（析构函数）时，我有时使用缩写形式 ctor 和 dtor。

Naming Conventions（命名惯例）

我试图为 objects（对象），classes（类），functions（函数），templates（模板）等选择意味深长的名字，但是在我的某些名字后面的含义可能不会立即显现出来。例如，我特别喜欢的两个 parameter names（参数名字）是 lhs 和 rhs。它们分别代表 "left-hand side" 和 "right-hand side"。我经常用它们作为实现 binary operators（二元运算符）的函数（例如，operator== 和 operator*）的 parameter names（参数名字）。例如，如果 a 和 b 是代表有理数的 objects（对象），而且如果 Rational objects（对象）能通过一个 non-member（非成员）的 operator* 函数相乘（Item 24 中解释的很可能就是这种情况），表达式

```
a * b
```

与函数调用

```
operator*(a,b)
```

就是等价的。

在 Item 24 中，我这样声明 operator*：

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

你可以看到，left-hand operand（左手操作数）a 在函数内部以 lhs 的面目出现，而 right-hand operand（右手操作数）b 以 rhs 的面目出现。

对于 member functions（成员函数），left-hand argument（左手参数）表现为 this pointer（this 指针），所以有时候我单独使用 parameter name（参数名字） rhs。你可能已经在第 5 页中某些 Widget 的 member functions（成员函数）的 declarations（声明）（本文介绍 copy constructor（拷贝构造函数）的那一段中的例子——译者注）中注意到了这一点。这一点提醒了我。我经常在示例中使用 Widget class（类）。"Widget" 并不意味着什么东西。它仅仅是在我需要一个示例类的名字的时候不时地使用一下的名字。它和 GUI 工具包中的 widgets 没有任何关系。

我经常遵循这个规则为 pointers（指针）命名：一个指向 type（类型）T 的 object（对象）的 pointer（指针）被称为 pt，"pointer to T"。以下是例子：

```
Widget *pw; // pw = ptr to Widget

class Airplane;

Airplane *pa; // pa = ptr to Airplane

class GameCharacter;

GameCharacter *pgc; // pgc = ptr to GameCharacter
```

我对 references（引用）使用类似的惯例：rw 可以认为是一个 reference to a

Widget（引向一个 Widget 的引用），而 ra 是一个 reference to an Airplane（引向一个 Airplane 的引用）。

在讨论 member functions（成员函数）的时候我偶尔会使用名字 mf。

Threading Considerations（对线程的考虑）

作为一种语言，C++ 没有 threads（线程）的概念——实际上，是没有任何一种 concurrency（并发）的概念。对于 C++ 标准库也是同样如此。就 C++ 涉及的范围而言，multithreaded programs（多线程编程）并不存在。

而且至今它们依然如此。我致力于让此书基于标准的，可移植的 C++，但我也不能对 thread safety（线程安全）已成为很多程序员所面临的一个问题的事实视而不见。我对付这个标准 C++ 和现实之间的裂痕的方法就是指出某个 C++ constructs（结构成分）以我的分析很可能在 threaded environment（线程环境）中引起问题的地方。这样不但不会使本书成为一本 multithreaded programming with C++（用 C++ 进行多线程编程）的书。反而，它更会使本书在相当程度上成为这样一本 C++ 编程的书：将自己在很大程度上限制于 single-threaded（单线程）思路，承认 multithreading（多线程）的存在，并试图指出有线程意识的程序员需要特别留心评估我提供的建议的地方。

如果你不熟悉 multithreading（多线程）或者不必为此担心，你可以忽略我关于线程的讨论。如果你正在编写一个多线程的应用或库，无论如何，请记住我的评注和并将它作为你使用 C++ 时需要致力去解决的问题的起点。

TR1 和 Boost

你会发现提及 TR1 和 Boost 的地方遍及全书。它们每一个都有一个专门的 Item 在某些细节上进行描述（Item 54 是 TR1，Item 55 是 Boost），但是，不幸的是，这些 Item 在全书的最后。（他们在那里是因为那样更好一些，我确实试过很多其它的地方。）如果你愿意，你现在就可以翻开并阅读那些 Item，但是如果你更喜欢从本书的起始处而不是结尾处开始，以下摘要会对你有所帮助：

- TR1 ("Technical Report 1") 是被加入 C++ 标准库的新机能的 specification（规格说明书）。这些机能以新的 class（类）和 function templates（函数模板）的形式提供了诸如 hash tables（哈希表），reference-counting smart pointers（引用计数智能指针），

regular expressions（正则表达式），等等。所有的 TR1 组件都位于嵌套在 namespace std 内部的 namespace tr1 内。

- Boost 是一个组织和一个网站 (http://boost.org) 提供的可移植的，经过同行评审的，开源的 C++ 库。大多数 TR1 机能都基于 Boost 的工作，而且直到编译器厂商在他们的 C++ 库发行版中包含 TR1 之前，Boost 网站很可能会保持开发者寻找 TR1 实现的第一站的地位。Boost 提供的东西比用于 TR1 的更多，无论如何，在很多情况下，它还是值得去了解一下的。

Item 1: 将 C++ 视为 federation of languages（语言联合体）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

最初，C++ 仅仅是在 C 的基础上附加了一些 object-oriented（面向对象）的特性。C++ 最初的名称—— "C with Classes" 就非常直观地表现了这一点。

作为一个语言的成熟过程，C++ 的成长大胆而充满冒险，它吸收的思想，特性，以至于编程策略与 C with Classes 越来越不同。exceptions（异常）要求不同的建构功能的途径（参见 Item 29），templates（模板）将设计思想提升到新的高度（参见 Item 41），而 STL 定义了一条前所未见的通向扩展性的道路。

今天的 C++ 已经成为一个 multiparadigm programming language（多范式的编程语言），一个囊括了 procedural（过程化），object-oriented（面向对象），functional（函数化），generic（泛型）以及 metaprogramming（元编程）特性的联合体。这些能力和弹性使 C++ 成为无可匹敌的工具，但也引起了一些混乱。所有的 "proper usage"（惯用法）规则似乎都有例外。我们该如何认识这样一个语言？

最简单的方法是不要将 C++ 视为一个单一的语言，而是一个亲族的语言的 federation（联合体）。在每一个特定的 sublanguage（子语言）中，它的特性趋向于直截了当，简单易记。但你从一个 sublanguage（子语言）转到另外一个，它的规则也许会发生变化。为了感受 C++，你必须将它的主要的 sublanguages（子语言）组织到一起。幸运的是，它只有 4 个：

- C ——归根结底，C++ 依然是基于 C 的。blocks（模块），statements（语句），preprocessor（预处理器），built-in data types（内建数据类型），arrays（数组），pointers（指针）等等，全都来自于 C。在很多方面。C++ 提出了比相应的 C 版本更高级的解决问题的方法（例如，参见 Item 2（选择 preprocessor（预处理器））和 13（使用 objects（对象）管理 resources（资源））），但是，当你发现你自己工作在 C++ 的 C 部分时，effective programming（高效编程）的规则表现了 C 的诸多限制范围：没有 templates（模板），没有 exceptions（异常），没有 overloading（重载）等等。
- Object-Oriented C++ —— C++ 的这部分就是 C with Classes 涉及到的全部：classes（类）（包括构造函数和析构函数），encapsulation（封装），inheritance（继承），polymorphism（多态），virtual functions (dynamic binding)（虚拟函数（动态绑定））等。C++ 的这一部分直接适用于 object-oriented design（面向对象设计）的经典规则。

- Template C++ ——这是 C++ 的 generic programming（泛型编程）部分，大多数程序员对此都缺乏经验。template（模板）的考虑已遍及 C++，而且好的编程规则中包含特殊的 template-only（模板专用）条款已不再不同寻常（参见 Item 46 通过调用 template functions（模板函数）简化 type conversions（类型转换））。实际上，templates（模板）极为强大，它提供了一种全新的 programming paradigm（编程范式）—— template metaprogramming (TMP) （模板元编程）。Item 48 提供了一个 TMP 的概述，但是，除非你是一个 hard-core template junkie（死心塌地的模板瘾君子），否则你不需在此费心，TMP 的规则对主流的 C++ 编程少有影响。

- STL —— STL 是一个 template library（模板库），但它一个非常特殊的 template library（模板库）。它将 containers（容器），iterators（迭代器），algorithms（算法）和 function objects（函数对象）非常优雅地整合在一起，但是。templates（模板）和 libraries（库）也可以围绕其它的想法建立起来。STL 有很多独特的处事方法，当你和 STL 一起工作，你需要遵循它的规则。

在头脑中保持这四种 sublanguages（子语言），当你从一种 sublanguage（子语言）转到另一种时，为了高效编程你需要改变你的策略，不要吃惊你遭遇到的情景。例如，使用 built-in（内建）（也就是说，C-like（类 C 的））类型时，pass-by-value（传值）通常比 pass-by-reference（传引用）更高效，但是当你从 C++ 的 C 部分转到 Object-Oriented C++（面向对象 C++），user-defined constructors（用户自定义构造函数）和 destructors（析构函数）意味着，通常情况下，更好的做法是 pass-by-reference-to-const（传引用给 const）。在 Template C++ 中工作时，这一点更加重要，因为，在这种情况下，你甚至不知道你的操作涉及到的 object（对象）的类型。然而，当你进入 STL，你知道 iterators（迭代器）和 function objects（函数对象）以 C 的 pointers（指针）为原型，对于 STL 中的 iterators（迭代器）和 function objects（函数对象），古老的 C 中的 pass-by-value（传值）规则又重新生效。（关于选择 parameter-passing（参数传递）方式的全部细节，参见 Item 20。）

C++ 不是使用一套规则的单一语言，而是 federation of four sublanguages（四种子语言的联合体），每一种都有各自的规则。在头脑中保持这些 sublanguages（子语言），你会发现对 C++ 的理解会容易得多。

Things to Remember

- effective C++ programming（高效 C++ 编程）规则的变化，依赖于你使用 C++ 的哪一个部分。

Item 2: 用 consts, enums 和 inlines 取代 #defines

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

这个 Item 改名为“用 compiler（编译器）取代 preprocessor（预处理器）”也许更好一些，因为 #define 根本就没有被看作是语言本身的一部分。这是它很多问题中的一个。当你像下面这样做：

```
#define ASPECT_RATIO 1.653
```

compiler（编译器）也许根本就没有看见这个符号名 ASPECT_RATIO，在 compiler（编译器）得到源代码之前，这个名字就已经被 preprocessor（预处理器）消除了。结果，名字 ASPECT_RATIO 可能就没有被加入 symbol table（符号表）。如果在编译的时候，发现一个 constant（常量）使用的错误，你可能会陷入混乱之中，因为错误信息中很可能用 1.653 取代了 ASPECT_RATIO。如果，ASPECT_RATIO 不是你写的，而是在头文件中定义的，你可能会对 1.653 的出处毫无头绪，你还会为了跟踪它而浪费时间。在 symbolic debugger（符号调试器）中也会遇到同样的问题，同样是因为这个名字可能并没有被加入 symbol table（符号表）。

解决方案是用 constant（常量）来取代 macro（宏）：

```
const double AspectRatio = 1.653;   // uppercase names are usually for
                                    // macros, hence the name change
```

作为一个 language constant（语言层面上的常量），AspectRatio 被 compilers（编译器）明确识别并确实加入 symbol table（符号表）。另外，对于 floating point constant（浮点常量）（比如本例）来说，使用 constant（常量）比使用 #define 能产生更小的代码。这是因为 preprocessor（预处理器）盲目地用 1.653 置换 macro name（宏名字）ASPECT_RATIO，导致你的 object code（目标代码）中存在多个 1.653 的拷贝，如果使用 constant（常量）AspectRatio，就绝不会产生多于一个的拷贝。

用 constant（常量）代替 #defines 时，有两个特殊情况值得提出。首先是关于 constant pointers（常量指针）的定义。因为 constant definitions（常量定义）通常被放在 header files（头文件）中（这样它们就可以被包含在多个 source files（源文件）中），除了

pointer（指针）指向的目标是常量外，pointer（指针）本身被声明为 const 更加重要。例如，在头文件中定义一个 constant char*-based string（基于 char* 的字符串常量）时，你必须写两次 const：

```
const char * const authorName = "Scott Meyers";
```

对于 const（特别是与 pointers（指针）相结合时）的意义和使用的完整讨论，请参见 Item 3。然而在此值的一提的是，string objects（对象）通常比它的 char*-based（基于 char*）的祖先更可取，所以，更好的 authorName 的定义方式如下：

```
const std::string authorName("Scott Meyers");
```

第二个特殊情况涉及到 class-specific constants（类属（类内部专用的）常量）。为了将一个 constant（常量）的作用范围限制在一个 class（类）内，你必须将它作为一个类的 member（成员），而且为了确保它最多只有一个 constant（常量）拷贝，你还必须把它声明为一个 static member（静态成员）。

```
class GamePlayer {

private:

  static const int NumTurns = 5;      // constant declaration

  int scores[NumTurns];               // use of constant
  ...
};
```

你从上面只看到了 NumTurns 的 declaration（声明），而不是 definition（定义）。通常，C++ 要求你为你使用的任何东西都提供一个 definition（定义），但是一个 static（静态）的 integral type（整型族）（例如：integers（整型），chars，bools）的 class-specific constants（类属常量）是一个例外。只要你不去取得它们的 address（地址），你可以只声明并使用它，而不提供它的 definition（定义）。如果你要取得一个 class constant（类属常量）的 address（地址），或者你使用的 compiler（编译器）在你没有取得 address（地址）时也不正确地要求 definition（定义）的话，你可以提供如下这样一个独立的 definition（定义）：

```
const int GamePlayer::NumTurns;     // definition of NumTurns; see
                                    // below for why no value is given
```

你应该把它放在一个 implementation file（实现文件）而非 header file（头文件）中。因为 class constants（类属常量）的 initial value（初始值）在声明时已经提供（例如：NumTurns 在定义时被初始化为 5），因此在定义处允许没有 initial value（初始值）。

注意，顺便提一下，没有办法使用 #define 来创建一个 class-specific constant（类属常量），因为 #defines 不考虑 scope（作用范围）。一旦一个 macro（宏）被定义，它将大范围影响你的代码（除非在后面某处存在 #undefed）。这就意味着，#defines 不仅不能用于 class-specific constants（类属常量），而且不能提供任何形式的 encapsulation（封装），也就是说，没有类似 "private"（私有）#define 的东西。当然，const data members（const 数据成员）是能够被封装的，NumTurns 就是如此。

比较老的 compilers（编译器）可能不接受上面的语法，因为它习惯于将一个 static class member（静态类成员）在声明时就获得 initial value（初始值）视为非法。而且，in-class initialization （类内初始化）仅仅对于 integral types（整型族）和 constants（常量）才被允许。如果上述语法不能使用，你可以将 initial value（初始值）放在定义处：

```
class CostEstimate {

private:

  static const double FudgeFactor;   // declaration of static class
  ...                                // constant; goes in header file
};

const double                         // definition of static class

  CostEstimate::FudgeFactor = 1.35;  // constant; goes in impl. file
```

这就是你所要做的全部。仅有的例外是当在类的编译期需要 value of a class constant（一个类属常量的值）的情况，例如前面在声明 array（数组）GamePlayer::scores 时（compilers（编译器）必须在编译期知道 array（数组）的 size（大小））。如果 compilers（编译器）（不正确地）禁止这种关于 static integral class constants（静态整型族类属常量）的 initial values（初始值）的使用方法的 in-class specification（规范），一个可接受的替代方案被亲切地（并非轻蔑地）昵称为 "the enum hack"。这项技术获得以下事实的支持：一个 enumerated type（枚举类型）的值可以用在一个需要 ints 的地方。所以 GamePlayer 可以被如下定义：

```
class GamePlayer {

private:

  enum { NumTurns = 5 };             // "the enum hack" - makes
                                     // NumTurns a symbolic name for 5

  int scores[NumTurns];              // fine
  ...
};
```

the enum hack 有几个值得被人所知的原因。首先，the enum hack 的行为在几个方面上更像一个 #define 而不是 const，而有时这正是你所需要的。例如：可以合法地取得一个 const 的 address（地址），但不能合法地取得一个 enum 的 address（地址），这正像同样不能合法地取得一个 #define 的 address（地址）。如果你不希望人们得到你的 integral constants（整型族常量）的 pointer（指针）或 reference（引用），enum（枚举）就是强制约束这一点的

好方法。（关于更多的通过编码的方法强制执行设计约束的方法，参见 Item 18。）同样，使用好的 compilers（编译器）不会为 integral types（整型族类型）的 const objects（const 对象）分配多余的内存（除非你创建了这个对象的指针或引用），即使拖泥带水的 compilers（编译器）乐意，你也决不会乐意为这样的 objects（对象）分配多余的内存。像 #defines 和 enums（枚举）就绝不会导致这种 unnecessary memory allocation（不必要的内存分配）。

需要知道 the enum hack 的第二个理由是纯粹实用主义的，大量的代码在使用它，所以当你看到它时，你要认识它。实际上，the enum hack 是 template metaprogramming（模板元编程）的一项基本技术（参见 Item 48）。

回到 preprocessor（预处理器）上来，#define 指令的另一个普遍的（不好的）用法是实现看来像函数，但不会引起一个函数调用的开销的 macros（宏）。以下是一个用较大的宏参数调用函数 f 的 macro（宏）：

```
// call f with the maximum of a and b

#define CALL_WITH_MAX(a, b) f((a) &gt; (b) ? (a) : (b))
```

这样的 macro（宏）有数不清的缺点，想起来就让人头疼。

无论何时，你写这样一个 macro（宏），都必须记住为 macro body（宏体）中所有的 arguments（参数）加上括号。否则，当其他人在表达式中调用了你的 macro（宏），你将陷入麻烦。但是，即使你确实做到了这一点，你还是会看到意想不到的事情发生：

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b); // a is incremented twice

CALL_WITH_MAX(++a, b+10); // a is incremented once
```

这里，调用 f 之前 a 递增的次数取决于它和什么进行比较！

幸运的是，你并不是必须要和这样不知所云的东西打交道。你可以通过一个 inline function（内联函数）的 template（模板）来获得 macro（宏）的效率，以及完全可预测的行为和常规函数的类型安全（参见 Item 30）：

```
template<typename T>                               // because we don't

inline void callWithMax(const T& a, const T& b)    // know what T is, we

{                                                  // pass by reference-to-

  f(a > b ? a : b);                                // const - see Item 20

}
```

这个 template（模板）产生一组函数，每一个获得两个相同类型的对象并使用其中较大的一个调用 f。这样就不需要为函数体内部的参数加上括号，也不需要担心多余的参数解析次数，等等。此外，因为 callWithMax 是一个真正的函数，它遵循函数的作用范围和访问规则。例如，谈论一个类的私有的 inline function（内联函数）会获得正确的理解，但是用 macro（宏）就无法做到这一点。

为了得到 consts，enums 和 inlines 的可用性，你需要尽量减少 preprocessor（预处理器）（特别是 #define）的使用，但还不能完全消除。#include 依然是基本要素，而 #ifdef/#ifndef 也继续扮演着重要的角色。现在还不是让 preprocessor（预处理器）完全退休的时间，但你应该给它漫长而频繁的假期。

Things to Remember

- 对于 simple constants（简单常量），用 const objects（const 对象）或 enums（枚举）取代 #defines。
- 对于 function-like macros（类似函数的宏），用 inline functions（内联函数）取代 #defines。

Item 3: 只要可能就用 const

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

关于 const 的一件美妙的事情是它允许你指定一种 semantic（语义上的）约束：一个特定的 object（对象）不应该被修改。而 compilers（编译器）将执行这一约束。它允许你通知 compilers（编译器）和其他程序员，某个值应该保持不变。如果确实如此，你就应该明确地表示出来，因为这样一来，你就可以谋求 compilers（编译器）的帮助，确保这个值不会被改变。

keyword（关键字）const 非常多才多艺。在 classes（类）的外部，你可以将它用于 global（全局）或 namespace（命名空间）范围的 constants（常量）（参见 Item 2），以及那些在 file（文件）、function（函数）或 block（模块）scope（范围）内被声明为 static（静态）的对象。在 classes（类）的内部，你可以将它用于 static（静态）和 non-static（非静态）data members（数据成员）上。对于 pointers（指针），你可以指定这个 pointer（指针）本身是 const，或者它所指向的数据是 const，或者两者都是，或者都不是：

```
char greeting[] = "Hello";

char *p = greeting; // non-const pointer,
                    // non-const data

const char *p = greeting; // non-const pointer,
                          // const data

char * const p = greeting; // const pointer,
                           // non-const data

const char * const p = greeting; // const pointer,
                                 // const data
```

这样的语法本身其实并不像表面上那样反复无常。如果 const 出现在星号左边，则指针 pointed to（指向）的内容为 constant（常量）；如果 const 出现在星号右边，则 pointer itself（指针自身）为 constant（常量）；如果 const 出现在星号两边，则两者都为 constant（常量）。

当指针指向的内容为 constant（常量）时，一些人将 const 放在类型之前，另一些人将它放在类型之后星号之前。两者在意义上并没有区别，所以，如下两个函数具有相同的 parameter type（参数类型）：

```
void f1(const Widget *pw); // f1 takes a pointer to a
                           // constant Widget object

void f2(Widget const *pw); // so does f2
```

因为它们都存在于实际的代码中，你应该习惯于这两种形式。

STL iterators（迭代器）以 pointers（指针）为原型，所以一个 iterator 在行为上非常类似于一个 T* pointer（指针）。声明一个 iterator 为 const 就类似于声明一个 pointer（指针）为 const（也就是说，声明一个 T* const pointer（指针））：不能将这个 iterator 指向另外一件不同的东西，但是它所指向的东西本身可以变化。如果你要一个 iterator 指向一个不能变化的东西（也就是一个 const T* pointer（指针）的 STL 对等物），你需要一个 const_iterator：

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =     // iter acts like a T* const
  vec.begin();
*iter = 10;                                 // OK, changes what iter points to
++iter;                                     // error! iter is const

std::vector<int>::const_iterator cIter =    // cIter acts like a const T*
  vec.begin();
*cIter = 10;                                // error! *cIter is const
++cIter;                                    // fine, changes cIter
```

对 const 最强有力的用法来自于它在 function declarations（函数声明）中的应用。在一个 function declaration（函数声明）中，const 既可以用在函数的 return value（返回值）上，也可以用在个别的 parameters（参数）上，对于 member functions（成员函数），还可以用于整个函数。

一个函数返回一个 constant value（常量值），常常可以在不放弃安全和效率的前提下尽可能减少客户的错误造成的影响。例如，考虑在 Item 24 中考察的 rational numbers（有理数）的 operator* 函数的声明。

```
class Rational { ... };

const Rational operator*(const Rational& lhs, const Rational& rhs);
```

很多第一次看到这些的程序员会不以为然。为什么 operator* 的结果应该是一个 const object（对象）？因为如果它不是，客户就可以犯下如此暴行：

```
Rational a, b, c;

...

(a * b) = c; // invoke operator= on the
             // result of a*b!
```

我不知道为什么一些程序员要为两个数的乘积赋值，但是我知道很多程序员这样做也并非不称职。所有这些可能来自一个简单的输入错误（要求这个类型能够隐式转型到 bool）：

```
if (a * b = c) ... // oops, meant to do a comparison!
```

如果 a 和 b 是 built-in type（内建类型），这样的代码显而易见是非法的。一个好的 user-defined types（用户自定义类型）的特点就是要避免与 built-ins（内建类型）毫无理由的不和谐（参见 Item 18），而且对我来说允许给两个数的乘积赋值看上去正是毫无理由的。将 operator* 的返回值声明为 const 就可以避免这一点，这就是我们要这样做的理由。

关于 const parameters（参数）没什么特别新鲜之处——它们的行为就像 local（局部）的 const objects（对象），而且无论何时，只要你能，你就应该这样使用。除非你需要改变一个 parameter（参数）或 local object（本地对象）的能力，否则，确保将它声明为 const。它只需要你键入六个字符，就能将你从我们刚刚看到的这个恼人的错误中拯救出来："我想键入 '==', 但我意外地键入了 '='"。

const member functions（const 成员函数）

member functions（成员函数）被声明为 const 的目的是标明这个 member functions（成员函数）可能会被 const objects（对象）调用。因为两个原因，这样的 member functions（成员函数）非常重要。首先，它使一个 class（类）的 interface（接口）更容易被理解。知道哪个函数可以改变 object（对象）而哪个不可以是很重要的。第二，它们可以和 const objects（对象）一起工作。因为，书写高效代码有一个很重要的方面，就像 Item 20 所解释的，提升一个 C++ 程序的性能的基本方法就是 pass objects by reference-to-const（以传引用给 const 的方式传递一个对象）。这个技术只有在 const member functions（成员函数）和作为操作结果的 const-qualified objects（被 const 修饰的对象）存在时才是可行的。

很多人没有注意到这样的事实，即 member functions（成员函数）在只有 constness（常量性）不同时是可以被 overloaded（重载）的，但这是 C++ 的一个重要特性。考虑一个代表文本块的类：

```
class TextBlock {
public:
  ...
  const char& operator[](std::size_t position) const   // operator[] for
  { return text[position]; }                           // const objects

  char& operator[](std::size_t position)               // operator[] for
  { return text[position]; }                           // non-const objects

private:
   std::string text;
};
```

TextBlock 的 operator[]s 可能会这样使用：

```
TextBlock tb("Hello");
std::cout << tb[0];                    // calls non-const
                                       // TextBlock::operator[]
const TextBlock ctb("World");
std::cout << ctb[0];                   // calls const TextBlock::operator[]
```

顺便提一下，const objects（对象）在实际程序中最经常出现的是作为这样一个操作的结果：passed by pointer- or reference-to-const（以传指针或者引用给 const 的方式传递）。上面的 ctb 的例子是人工假造的。下面这个例子更真实一些：

```
void print(const TextBlock& ctb)       // in this function, ctb is const
{
  std::cout << ctb[0];                 // calls const TextBlock::operator[]
  ...
}
```

通过 overloading（重载） operator[]，而且给不同的版本不同的返回类型，你能对 const 和 non-const 的 TextBlocks 做不同的操作：

```
std::cout << tb[0];                    // fine — reading a
                                       // non-const TextBlock

tb[0] = 'x';                           // fine — writing a
                                       // non-const TextBlock

std::cout << ctb[0];                   // fine — reading a
                                       // const TextBlock

ctb[0] = 'x';                          // error! — writing a
                                       // const TextBlock
```

请注意这里的错误只与被调用的 operator[] 的 return type（返回类型）有关，而调用 operator[] 本身总是正确的。错误出现在企图为 const char& 赋值的时候，因为它是 const 版本的 operator[] 的 return type（返回类型）。

再请注意 non-const 版本的 operator[] 的 return type（返回类型）是 reference to a char（一个 char 的引用）而不是一个 char 本身。如果 operator[] 只是返回一个简单的 char，下面的语句将无法编译：

```
tb[0] = 'x';
```

因为改变一个返回 built-in type（内建类型）的函数的返回值总是非法的。即使它合法，C++ returns objects by value（以传值方式返回对象）这一事实（参见 Item 20）也意味着被改变的是 tb.text[0] 的一个 copy（拷贝），而不是 tb.text[0] 自己，这不会是你想要的行为。

让我们为哲学留一点时间。看看一个 member function（成员函数）是 const 意味着什么？有两个主要的概念：bitwise constness（二进制位常量性）（也称为 physical constness（物理常量性））和 logical constness（逻辑常量性）。

bitwise（二进制位）const 派别坚持认为，一个 member function（成员函数），当且仅当它不改变 object（对象）的任何 data members（数据成员）（static（静态的）除外），也就是说如果不改变 object（对象）内的任何 bits（二进制位），则这个 member function（成员函数）就是 const。bitwise constness（二进制位常量性）的一个好处是比较容易监测违例：编译器只需要寻找对 data members（数据成员）的 assignments（赋值）。实际上，bitwise

constness（二进制位常量性）就是 C++ 对 constness（常量性）的定义，一个 const member function（成员函数）不被允许改变调用它的 object（对象）的任何 non-static data members（非静态数据成员）。

不幸的是，很多效果上并不是完全 const 的 member functions（成员函数）通过了 bitwise（二进制位）的检验。特别是，一个经常改变某个 pointer（指针）指向的内容的 member function（成员函数）效果上不是 const 的。除非这个 pointer（指针）在这个 object（对象）中，否则这个函数就是 bitwise（二进制位）const 的，编译器也不会提出异议。例如，假设我们有一个 TextBlock-like class（类似 TextBlock 的类），因为它需要与一个不知 string objects（对象）为何物的 C API 打交道，所以它需要将它的数据存储为 char* 而不是 string。

```
class CTextBlock {
public:
  ...

  char& operator[](std::size_t position) const   // inappropriate (but bitwise

  { return pText[position]; }                     // const) declaration of
                                                  // operator[]
private:
  char *pText;
};
```

尽管 operator[] 返回 a reference to the object's internal data（一个引向对象内部数据的引用），这个 class（类）还是（不适当地）将它声明为一个 const member function（成员函数）（Item 28 将谈论一个深入的主题）。先将它放到一边，看看 operator[] 的实现，它并没有使用任何手段改变 pText。结果，编译器愉快地生成了 operator[] 的代码，因为毕竟对所有编译器而言，它都是 bitwise（二进制位）const 的，但是我们看看会发生什么：

```
const CTextBlock cctb("Hello");        // declare constant object

char *pc = &cctb[0];                   // call the const operator[] to get a
                                       // pointer to cctb's data

*pc = 'J';                             // cctb now has the value "Jello"
```

这里确实出了问题，你用一个 particular value（确定值）创建一个 constant object（常量对象），然后你只是用它调用了 const member functions（成员函数），但是你还是改变了它的值！

这就引出了 logical constness（逻辑常量性）的概念。这一理论的信徒认为：一个 const member function（成员函数）可能会改变调用它的 object（对象）中的一些 bits（二进制位），但是只能用客户无法察觉的方法。例如，你的 CTextBlock class（类）在需要的时候可以储存文本块的长度：

```
class CTextBlock {
public:

  ...

  std::size_t length() const;

private:
  char *pText;
  std::size_t textLength;             // last calculated length of textblock
  bool lengthIsValid;                 // whether length is currently valid
};

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);  // error! can't assign to textLength
    lengthIsValid = true;             // and lengthIsValid in a const
  }                                   // member function

  return textLength;
}
```

length 的实现当然不是 bitwise（二进制位）const 的—— textLength 和 lengthIsValid 都可能会被改变——但是它还是被看作对 const CTextBlock 对象有效。但编译器不同意，它还是坚持 bitwise constness（二进制位常量性），怎么办呢？

解决方法很简单：利用以 mutable 闻名的 C++ 的 const-related（const 相关）的灵活空间。mutable 将 non-static data members（非静态数据成员）从 bitwise constness（二进制位常量性）的约束中解放出来：

```
class CTextBlock {
public:

  ...

  std::size_t length() const;

private:
  char *pText;

  mutable std::size_t textLength;         // these data members may
  mutable bool lengthIsValid;             // always be modified, even in
};                                        // const member functions

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);      // now fine
    lengthIsValid = true;                 // also fine
  }

  return textLength;
}
```

避免 const 和 non-const member functions（成员函数）的重复

mutable 对于解决 bitwise-constness-is-not-what-I-had-in-mind（二进制位常量性不太合我的心意）的问题是一个不错的解决方案，但它不能解决全部的 const-related（const 相关）难题。例如，假设 TextBlock（包括 CTextBlock）中的 operator[] 不仅要返回一个适当的字符的

reference（引用），它还要进行 bounds checking（边界检查），logged access information（记录访问信息），甚至 data integrity validation（数据完整性确认），将这些功能都加入到 const 和 non-const 的 operator[] 函数中（不必为我们现在有着非凡长度的 implicitly inline functions（隐含内联函数）而烦恼，参见 Item 30），使它们变成如下这样的庞然大物：

```
class TextBlock {
public:

  ...

  const char& operator[](std::size_t position) const
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data integrity
    return text[position];
  }

  char& operator[](std::size_t position)
  {
    ...                                 // do bounds checking
    ...                                 // log access data
    ...                                 // verify data integrity
    return text[position];
  }

private:
   std::string text;
};
```

哎呀！你是说 code duplication（重复代码）？还有随之而来的额外的编译时间，维护成本以及代码膨胀等令人头痛之类的事情吗？当然，也可以将 bounds checking（边界检查）等全部代码转移到一个单独的 member function（成员函数）（自然是 private（私有）的）中，并让两个版本的 operator[] 来调用它，但是，你还是要重复写出调用那个函数和 return 语句的代码。

你真正要做的是只实现一次 operator[] 的功能，而使用两次。换句话说，你可以用一个版本的 operator[] 去调用另一个版本。并可以为我们 casting away（通过强制转型脱掉）constness（常量性）。

作为一个通用规则，casting（强制转型）是一个非常坏的主意，我会投入整个一个 Item 的篇幅来告诉你不要使用它（Item 27），但是 code duplication（重复代码）也不是什么好事。在当前情况下，const 版本的 operator[] 所做的事也正是 non-const 版本所做的，仅有的不同是它有一个 const-qualified return type（被 const 修饰的返回类型）。在这种情况下，casting away（通过强制转型脱掉）return value（返回类型）的 const 是安全的，因为，无论谁调用 non-const operator[]，首先要有一个 non-const object（对象）。否则，它不能调用一个 non-const 函数。所以，即使需要一个 cast（强制转型），让 non-const operator[] 调用 const 版本也是避免重复代码的安全方法。代码如下，你读了后面的解释后对它的理解可能会更加清晰：

```
class TextBlock {
public:

  ...

  const char& operator[](std::size_t position) const     // same as before
  {
    ...
    ...
    ...
    return text[position];
  }

  char& operator[](std::size_t position)         // now just calls const op[]
  {
    return
      const_cast<char&>(                         // cast away const on
                                                 // op[]'s return type;
        static_cast<const TextBlock&>(*this)     // add const to *this's type;
          [position]                             // call const version of op[]
      );
  }

...

};
```

正如你看到的，代码中有两处 casts（强制转型），而不是一处。我们让 non-const operator[] 调用 const 版本，但是，如果在 non-const operator[] 的内部，我们仅仅是调用 operator[]，那我们将递归调用我们自己。它会进行一百万次甚至更多。为了避免 infinite recursion（无限递归），我们必须明确指出我们要调用 const operator[]，但是没有直接的办法能做到这一点，于是我们将 *this* 从 *TextBlock&* 的自然类型强制转型到 *const TextBlock&*。是的，我们使用 *cast*（强制转型）为它加上了 *const*！所以我们有两次 *casts*（强制转型）：第一次是为 this 加上 const（以便在我们调用 operator[] 时调用它的 const 版本），第二次是从 const operator[] 的 return value（返回值）之中去掉 const。

加上 const 的 cast（强制转型）仅仅是强制施加一次安全的转换（从一个 non-const object（对象）到一个 const object（对象）），所以我们用一个 static_cast 来做。去掉 const 只能经由 const_cast 来完成，所以在这里我们没有别的选择。（在技术上，我们有。一个 C-style cast（C 风格的强制转型）也能工作，但是，就像我在 Item 27 中解释的，这样的 casts（强制转型）很少是一个正确的选择。如果你不熟悉 static_cast 或 const_cast，Item 27 中包含有一个概述。）

在完成其它事情的基础上，我们在此例中调用了一个 operator（操作符），所以，语法看上去有些奇怪。导致其不会赢得选美比赛，但是它根据 const 版本的 operator[] 实现其 non-const 版本而避免 code duplication（代码重复）的方法达到了预期的效果。使用丑陋的语法达到目标是否值得最好由你自己决定，但是这种根据 const member function（成员函数）实现它的 non-const 版本的技术却非常值得掌握。

更加值得掌握的是做这件事的反向方法——通过用 const 版本调用 non-const 版本来避免重复——是你不能做的。记住，一个 const member function（成员函数）承诺绝不会改变它的 object（对象）的逻辑状态，但是一个 non-const member function（成员函数）不会做这样

的承诺。如果你从一个 const member function（成员函数）调用一个 non-const member function（成员函数），你将面临你承诺不会变化的 object（对象）被改变的风险。这就是为什么使用一个 const member function（成员函数）调用一个 non-const member function（成员函数）是错误的，object（对象）可能会被改变。实际上，那样的代码如果想通过编译，你必须用一个 const_cast 来去掉 *this* 的 *const*，这是一个显而易见的麻烦。而反向的调用——就像我在上面用的——是安全的：一个 *non-const member function*（成员函数）对一个 *object*（对象）能够为所欲为，所以调用一个 *const member function*（成员函数）也没有任何风险。这就是为什么 *static_cast* 在这种情况下可以工作在 this 上的原因：这里没有 const-related 危险。

就像在本 Item 开始我所说的，const 是一件美妙的东西。在 pointers（指针）和 iterators（迭代器）上，在 pointers（指针），iterators（迭代器）和 references（引用）涉及到的 object（对象）上，在 function parameters（函数参数）和 return types（返回值）上，在 local variables（局部变量）上，在 member functions（成员函数）上，const 是一个强有力的盟友。只要可能就用它，你会为你所做的感到高兴。

Things to Remember

- 将某些东西声明为 const 有助于编译器发现使用错误。const 能被用于任何 scope（范围）中的 object（对象），用于 function parameters（函数参数）和 return types（返回类型），用于整个 member functions（成员函数）。
- 编译器坚持 bitwise constness（二进制位常量性），但是你应该用 conceptual constness（概念上的常量性）来编程。（此处原文有误，conceptual constness为作者在本书第二版中对 logical constness 的称呼，正文中的称呼改了，此处却没有改。其实此处还是作者新加的部分，却使用了旧的术语，怪！——译者注）
- 当 const 和 non-const member functions（成员函数）具有本质上相同的实现的时候，使用 non-const 版本调用 const 版本可以避免 code duplication（代码重复）。

Item 4: 确保 objects（对象）在使用前被初始化

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 看上去在对象的值的初始化方面变化莫测。例如，如果你这样做，

```
int x;
```

在某些情形下，x 会被初始化（为 0），但是在其它情形下，也可能没有。如果你这样做，

```
class Point {
  int x, y;
};

...

Point p;
```

p 的 data members（数据成员）有时会被初始化（为 0），但有时没有。如果你从一个不存在 uninitialized objects（未初始化对象）的语言来到 C++，请注意这个问题，因为它非常重要。

读取一个 uninitialized values（未初始化值）会引起 undefined behavior（未定义行为）。在一些平台上，读一个 uninitialized value（未初始化值）会引起程序中止，更可能的情况是得到一个你所读的那个位置上的 semi-random bits（半随机二进制位），最终导致不可预测的程序行为和恼人的调试。

现在，有一些描述关于什么时候能保证 object initialization（对象初始化）会发生什么时候不能保证的规则。不幸的是，这些规则很复杂——我觉得它复杂得无法记住。通常，如果你使用 C++ 的 C 部分（参见 Item 1），而且 initialization（初始化）可能会花费一些运行时间，它就不能保证发生。如果你使用 C++ 的 non-C 部分，事情会有些变化。这就是为什么一个 array（数组）（来自 C++ 的 C 部分）不能确保它的元素被初始化，但是一个 vector（来自 C++ 的 STL 部分）就能够确保。

处理这种事情的表面不确定状态的最好方法就是总是在使用之前初始化你的对象。对于 built-in types（内建类型）的 non-member objects（非成员对象），需要你手动做这件事。例如：

```
int x = 0;                              // manual initialization of an int

const char * text = "A C-style string";   // manual initialization of a
                                          // pointer (see also Item 3)

double d;                               // "initialization" by reading from

std::cin >> d;                          // an input stream
```

除此之外的几乎全部情况，initialization（初始化）的重任就落到了 constructors（构造函数）的身上。这里的规则很简单：确保 all constructors（所有的构造函数）都初始化了 object（对象）中的每一样东西。

这个规则很容易遵守，但重要的是不要把 assignment（赋值）和 initialization（初始化）搞混。考虑下面这个表现一个通讯录条目的 class（类）的 constructor（构造函数）：

```
class PhoneNumber { ... };

class ABEntry {                         // ABEntry = "Address Book Entry"

public:

  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);

private:

  std::string theName;

  std::string theAddress;

  std::list<PhoneNumber> thePhones;

  int num TimesConsulted;

};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)

{

  theName = name;                       // these are all assignments,

  theAddress = address;                 // not initializations

  thePhones = phones;

  numTimesConsulted = 0;

}
```

这样做虽然使得 ABEntry objects（对象）具有了你所期待的值，但还不是最好的做法。C++ 的规则规定一个 object（对象）的 data members（数据成员）在进入 constructor（构造函数）的函数体之前被初始化。在 ABEntry 的 constructor（构造函数）内，theName，theAddress 和 thePhones 不是 being initialized（被初始化），而是 being assigned（被赋值）。initialization（初始化）发生得更早——在进入 ABEntry 的 constructor（构造函数）的

函数体之前，它们的 default constructors（缺省的构造函数）已经被自动调用。但不包括 numTimesConsulted，因为它是一个 built-in type（内建类型）。不能保证它在被赋值之前被初始化。

一个更好的写 ABEntry constructor（构造函数）的方法是用 member initialization list（成员初始化列表）来代替 assignments（赋值）：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)

: theName(name),
  theAddress(address),                  // these are now all initializations
  thePhones(phones),
  numTimesConsulted(0)

{}                                      // the ctor body is now empty
```

这个 constructor（构造函数）的最终结果和前面那个相同，但是通常它有更高的效率。assignment-based（基于赋值）的版本会首先调用 default constructors（缺省构造函数）初始化 theName，theAddress 和 thePhones，然而很快又在 default-constructed（缺省构造）的值之上赋予新值。那些 default constructions（缺省构造函数）所做的工作被浪费了。而 member initialization list（成员初始化列表）的方法避免了这个问题，因为 initialization list（初始化列表）中的 arguments（参数）就可以作为各种 data members（数据成员）的 constructor（构造函数）所使用的 arguments（参数）。在这种情况下，theName 从 name 中 copy-constructed（拷贝构造），theAddress 从 address 中 copy-constructed（拷贝构造），thePhones 从 phones 中 copy-constructed（拷贝构造）。对于大多数类型来说，只调用一次 copy constructor（拷贝构造函数）的效率比先调用一次 default constructor（缺省构造函数）再调用一次 copy assignment operator（拷贝赋值运算符）的效率要高（有时会高很多）。

对于 numTimesConsulted 这样的 built-in type（内建类型）的 objects（对象），initialization（初始化）和 assignment（赋值）没有什么不同，但为了统一性，最好是经由 member initialization（成员初始化）来 initialize（初始化）每一件东西。类似地，当你只想 default-construct（缺省构造）一个 data member（数据成员）时也可以使用 member initialization list（成员初始化列表），只是不必指定 initialization argument（初始化参数）而已。例如，如果 ABEntry 有一个不取得 parameters（参数）的 constructor（构造函数），它可以像这样实现：

```
ABEntry::ABEntry()
:theName(),                         // call theName's default ctor;
 theAddress(),                      // do the same for theAddress;
 thePhones(),                       // and for thePhones;
 numTimesConsulted(0)               // but explicitly initialize

{}                                  // numTimesConsulted to zero
```

因为对于那些在 member initialization list（成员初始化列表）中的，没有 initializers（初始化器）的，user-defined types（用户自定义类型）的 data members（数据成员），编译器会自动调用其 default constructors（缺省构造函数），所以一些程序员会认为上面的方法有些过分。这也不难理解，但是一个方针是：在 initialization list（初始化列表）中总是列出每一个 data member（数据成员），这就可以避免一旦发生疏漏就必须回忆起可能是哪一个 data members（数据成员）没有被初始化。例如，因为 numTimesConsulted 是一个 built-in type（内建类型），如果将它从 member initialization list（成员初始化列表）中删除，就为 undefined behavior（未定义行为）打开了方便之门。

有时，即使是 built-in types（内建类型），initialization list（初始化列表）也必须使用。比如，const 或 references（引用）data members（数据成员）是必须 be initialized（被初始化）的，它们不能 be assigned（被赋值）（参见 Item 5）。为了避免记忆什么时候 data members（数据成员）必须在 member initialization list（成员初始化列表）中初始化，而什么时候又是可选的，最简单的方法就是总是使用 initialization list（初始化列表）。它有时是必须的，而且它通常都比 assignments（赋值）更有效率。

很多 classes（类）有多个 constructors（构造函数），而每一个 constructor（构造函数）都有自己的 member initialization list（成员初始化列表）。如果有很多 data members（数据成员）和/或 base classes（基类），成倍增加的 initialization lists（初始化列表）的存在引起令人郁闷的重复（在列表中）和厌烦（在程序员中）。在这种情况下，不能不讲道理地从列表中删除那些 assignment（赋值）和 true initialization（真正的初始化）一样工作的 data members（数据成员）项目，而是将 assignments（赋值）移到一个单独的（当然是 private（私有）的）函数中，以供所有 constructors（构造函数）调用。这个方法对于那些 true initial values（真正的初始值）是从文件中读入或从数据库中检索出来的 data members（数据成员）特别有帮助。然而，通常情况下，true member initialization（真正的成员初始化）（经由一个 initialization list（初始化列表））比经由 assignment（赋值）来进行的 pseudo-initialization（假初始化）更可取。

C++ 并非变幻莫测的方面是一个 object（对象）的数据被初始化的顺序。这个顺序总是相同的：base classes（基类）在 derived classes（派生类）之前被初始化（参见 Item 12），在一个 class（类）内部，data members（数据成员）按照它们被声明的顺序被初始化。例如，在 ABEntry 中，theName 总是首先被初始化，theAddress 是第二个，thePhones 第三，numTimesConsulted 最后。即使它们在 member initialization list（成员初始化列表）中以一种不同的顺序排列（这不幸合法），这依然是成立的。为了避免读者混淆，以及一些模糊不清的行为引起错误的可能性，initialization list（初始化列表）中的 members（成员）的排列顺序应该总是与它们在 class（类）中被声明的顺序保持一致。

一旦处理了 built-in types（内建类型）的 non-member objects（非成员对象）的显式初始化，而且确保你的 constructors（构造函数）使用 member initialization list（成员初始化列表）初始化了它的 base classes（基类）和 data members（数据成员），那就只剩下一件事情需要费心了。那就是——深呼吸先——定义在不同 translation units（转换单元）中的 non-local static objects（非局部静态对象）的 initialization（初始化）的顺序。

让我们一片一片地把这个词组拆开。

一个 static object（静态对象）的生存期是从它创建开始直到程序结束。stack and heap-based objects（基于堆栈的对象）就被排除在外了。包括 global objects（全局对象），objects defined at namespace scope（定义在命名空间范围内的对象），objects declared static inside classes（在类内部声明为静态的对象），objects declared static inside functions（在函数内部声明为静态的对象）和 objects declared static at file scope（在文件范围内被声明为静态的对象）。static objects inside functions（在函数内部的静态对象）以 local static objects（局部静态对象）（因为它局部于函数）为人所知，其它各种 static objects（静态对象）以 non-local static objects（非局部静态对象）为人所知。程序结束时 static objects（静态对象）会自动销毁，也就是当 main 停止执行时会自动调用它们的 destructors（析构函数）。

一个 translation unit（转换单元）是可以形成一个单独的 object file（目标文件）的 source code（源代码）。基本上是一个单独的 source file（源文件），再加上它全部的 #include 文件。

我们关心的问题是这样的：包括至少两个分别编译的 source files（源文件），每一个中都至少包含一个 non-local static object（非局部静态对象）（也就是说，global（全局）的，at namespace scope（命名空间范围）的，static in a class（类内）的或 at file scope（文件范围）的 object（对象））。实际的问题是这样的：如果其中一个 translation unit（转换单元）内的一个 non-local static object（非局部静态对象）的 initialization（初始化）用到另一个 translation unit（转换单元）内的non-local static object（非局部静态对象），它所用到的 object（对象）可能没有被初始化，因为 the relative order of initialization of non-local static objects defined in different translation units is undefined（定义在不同转换单元内的非局部静态对象的初始化的相对顺序是没有定义的）。

一个例子可以帮助我们。假设你有一个 FileSystem class（类），可以使 Internet 上的文件看起来就像在本地。因为你的 class（类）使得世界看起来好像只有一个单独的 file system（文件系统），你可以在 global（全局）或 namespace（命名空间）范围内创建一个专门的 object（对象）来代表这个单独的 file system（文件系统）：

```
class FileSystem {                    // from your library

public:
  ...
  std::size_t numDisks() const;       // one of many member functions
  ...
};

extern FileSystem tfs;                // object for clients to use;
                                      // "tfs" = "the file system"
```

一个 FileSystem object（对象）绝对是举足轻重的，所以在 theFileSystem object（对象）被创建之前就使用将会损失惨重。

现在假设一些客户为一个 file system（文件系统）中的目录创建了一个 class（类），他们的 class（类）使用了 theFileSystem object（对象）：

```
class Directory {                       // created by library client

public:

   Directory( params );
  ...
};

Directory::Directory( params )

{
  ...
  std::size_t disks = tfs.numDisks();   // use the tfs object
  ...
}
```

更进一步，假设这个客户决定为临时文件创建一个单独的 Directory object（对象）：

```
Directory tempDir( params ); // directory for temporary files
```

现在 initialization order（初始化顺序）的重要性变得明显了：除非 tfs 在 tempDir 之前初始化，否则，tempDir 的 constructor（构造函数）就会在 tfs 被初始化之前试图使用它。但是，tfs 和 tempDir 是被不同的人于不同的时间在不同的 source files（源文件）中创建的——它们是定义在不同 translation units（转换单元）中的 non-local static objects（非局部静态对象）。你怎么能确保 tfs 一定会在 tempDir 之前被初始化呢？

你不能。重申一遍，the relative order of initialization of non-local static objects defined in different translation units is undefined（定义在不同转换单元内的非局部静态对象的初始化的相对顺序是没有定义的）。这是有原因的。决定 non-local static objects（非局部静态对象）的"恰当的"初始化顺序是困难的，非常困难，以至于无法完成。在最常见的形式下——多个 translation units（转换单元）和 non-local static objects（非局部静态对象）通过 implicit template instantiations（隐式模板实例化）产生（这本身可能也是经由 implicit template instantiations（隐式模板实例化）引起的）——不仅不可能确定一个正确的 initialization（初始化）顺序，甚至不值得去寻找可能确定正确顺序的特殊情况。

幸运的是，一个小小的设计改变从根本上解决了这个问题。全部要做的就是将每一个 non-local static object（非局部静态对象）移到它自己的函数中，在那里它被声明为 static（静态）。这些函数返回它所包含的 objects（对象）的引用。客户可以调用这些函数来代替直接涉及那些 objects（对象）。换一种说法，就是用 local static objects（局部静态对象）取代 non-local static objects（非局部静态对象）。（aficionados of design patterns（设计模式迷们）会认出这是 Singleton 模式的通用实现）。

这个方法建立在 C++ 保证 local static objects（局部静态对象）的初始化发生在因为调用那个函数而第一次遇到那个 object（对象）的 definition（定义）时候。所以，如果你用调用返回 references to local static objects（局部静态对象的引用）的函数的方法取代直接访问 non-

local static objects（非局部静态对象）的方法，你将确保你取回的 references（引用）引向 initialized objects（已初始化的对象）。作为一份额外收获，如果你从不调用这样一个仿效 non-local static object（非局部静态对象）的函数，你就不会付出创建和销毁这个 object（对象）的代价，而一个 true non-local static objects（真正的非局部静态对象）则不会有这样的效果。

以下就是这项技术在 tfs 和 tempDir 上的应用：

```
class FileSystem { ... };           // as before

FileSystem& tfs()                   // this replaces the tfs object; it could be
{                                   // static in the FileSystem class
  static FileSystem fs;             // define and initialize a local static object
  return fs;                        // return a reference to it
}

class Directory { ... };            // as before

Directory::Directory( params )      // as before, except references to tfs are
{                                   // now to tfs()
  ...
  std::size_t disks = tfs().numDisks();
  ...
}

Directory& tempDir()                // this replaces the tempDir object; it
{                                   // could be static in the Directory class
  static Directory td;              // define/initialize local static object
  return td;                        // return reference to it
}
```

这个改良系统的客户依然可以按照他们已经习惯的方法编程，只是他们现在应该用 tfs() 和 tempDir() 来代替 tfs 和 tempDir。也就是说，他们应该使用返回 references to objects（对象引用）的函数来代替使用 objects themselves（对象自身）。

按照以下步骤来写 reference-returning functions（返回引用的函数）总是很简单：在第 1 行定义并初始化一个 local static object（局部静态对象），在第 2 行返回它。这样的简单使它们成为 inlining（内联化）的完美的候选者，特别是在它们被频繁调用的时候（参见 Item 30）。在另一方面，这些函数包含 static object（静态对象）的事实使它们在 multithreaded systems（多线程系统）中会出现问题。更进一步，任何种类的 non-const static object（非常量静态对象）—— local（局部）的或 non-local（非局部）的——在 multiple threads（多线程）存在的场合都会发生麻烦。解决这个麻烦的方法之一是在程序的 single-threaded（单线程）的启动部分手动调用所有的 reference-returning functions（返回引用的函数）。以此来避免 initialization-related（与初始化相关）的混乱环境。

当然，用 reference-returning functions（返回引用的函数）来防止 initialization order problems（初始化顺序问题）的想法首先依赖于你的 objects（对象）有一个合理的 initialization order（初始化顺序）。如果你有一个系统，其中 object A 必须在 object B 之前初始化，但是 A 的初始化又依赖于 B 已经被初始化，你将遇到问题，坦白地讲，你遇到大麻烦了。然而，如果你避开了这种病态的境遇，这里描述的方法会很好地为你服务，至少在 single-threaded applications（单线程应用）中是这样。

避免在初始化之前使用 objects（对象），你只需要做三件事。首先，手动初始化 built-in types（内建类型）的 non-member objects（非成员对象）。第二，使用 member initialization lists（成员初始化列表）初始化一个 object（对象）的所有部分。最后，在设计中绕过搞乱定义在分离的 translation units（转换单元）中的 non-local static objects（非局部静态对象）initialization order（初始化顺序）的不确定性。

Things to Remember

- 手动初始化 built-in type（内建类型）的 objects（对象），因为 C++ 只在某些时候才会自己初始化它们。
- 在 constructor（构造函数）中，用 member initialization list（成员初始化列表）代替函数体中的 assignment（赋值）。initialization list（初始化列表）中 data members（数据成员）的排列顺序要与它们在 class（类）中被声明的顺序相同。
- 通过用 local static objects（局部静态对象）代替 non-local static objects（非局部静态对象）来避免跨 translation units（转换单元）的 initialization order problems（初始化顺序问题）。

Item 5: 了解 C++ 为你偷偷地加上和调用了什么函数

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

一个 empty class（空类）什么时候将不再是 empty class（空类）？答案是当 C++ 搞定了它。如果你自己不声明一个 copy constructor（拷贝构造函数），一个 copy assignment operator（拷贝赋值运算符）和一个 destructor（析构函数），编译器就会为这些东西声明一个它自己的版本。此外，如果你自己根本没有声明 constructor（构造函数），编译器就会为你声明一个 default constructor（缺省构造函数）。所有这些函数都被声明为 public 和 inline（参见 Item 30）。作为结果，如果你写

```
class Empty{};
```

在本质上和你这样写是一样的：

```
class Empty {

public:

  Empty() { ... }                           // default constructor

  Empty(const Empty& rhs) { ... }           // copy constructor

  ~Empty() { ... }                          // destructor — see below
                                            // for whether it's virtual

  Empty& operator=(const Empty& rhs) { ... } // copy assignment operator

};
```

这些函数只有在它们被需要的时候才会生成，但是并不需要做太多的事情，就会用到它们。下面的代码会促使每一个函数生成：

```
Empty e1;                               // default constructor;
                                        // destructor

Empty e2(e1);                           // copy constructor

e2 = e1;                                // copy assignment operator
```

假设编译器为你写了这些函数，那么它们做些什么呢？default constructor（缺省构造函数）和 destructor（析构函数）主要是给编译器一个地方放置 "behind the scenes" code（“幕后”代码）的，诸如 base classes（基类）和 non-static data members（非静态数据成员）的

constructors（构造函数）和 destructor（析构函数）的调用。注意，生成的 destructor（析构函数）是 non-virtual（非虚拟）的（参见 Item 7），除非它所在的 class（类）是从一个 base class（基类）继承而来，而 base class（基类）自己声明了一个 virtual destructor（虚拟析构函数）（这种情况下，函数的 virtualness（虚拟性）来自 base class（基类））。

对于 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符），compiler-generated versions（编译器生成版本）只是简单地从 source object（源对象）拷贝每一个 non-static data member（非静态数据成员）到 target object（目标对象）。例如，考虑一个 NamedObject template（模板），它允许你将名字和类型为 T 的 objects（对象）联系起来的：

```
template<typename T>
class NamedObject {
public:
  NamedObject(const char *name, const T& value);
  NamedObject(const std::string& name, const T& value);

  ...

private:
  std::string nameValue;
  T objectValue;
};
```

因为 NamedObject 中声明了一个 constructors（构造函数），编译器就不会再生成一个 default constructor（缺省构造函数）。这一点很重要，它意味着如果你小心地设计一个 class（类），使它需要 constructor arguments（构造函数参数），你就不必顾虑编译器会不顾你的决定，轻率地增加一个不需要参数的 constructors（构造函数）。

NamedObject 既没有声明 copy constructor（拷贝构造函数）也没有声明 copy assignment operator（拷贝赋值运算符），所以编译器将生成这些函数（如果需要它们的话）。看，这就是 copy constructor（拷贝构造函数）的用法：

```
NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1);                 // calls copy constructor
```

编译器生成的 copy constructor（拷贝构造函数）一定会用 no1.nameValue 和 no1.objectValue 分别初始化 no2.nameValue 和 no2.objectValue。nameValue 的类型是 string，标准 string 类型有一个 copy constructor（拷贝构造函数），所以将通过以 no1.nameValue 作为参数调用 string 的 copy constructor（拷贝构造函数）初始化 no2.nameValue。而另一方面，NamedObject<int>::objectValue 的类型是 int（因为在这个 template instantiation（模板实例化）中 T 是 int），而 int 是 built-in type（内建类型），所以将通过拷贝 no1.objectValue 的每一个二进制位初始化 no2.objectValue。

编译器为 NamedObject<int> 生成的 copy assignment operator（拷贝赋值运算符）本质上也会有同样的行为，但是，通常情况下，只有在结果代码合法而且有一个合理的可理解的巧合时，compiler-generated（编译器生成）的 copy assignment operator（拷贝赋值运算符）才会有我所描述的行为方式。如果这两项检测中的任一项失败了，编译器将拒绝为你的 class（类）生成一个 operator=。

例如，假设 NamedObject 如下定义，nameValue 是一个 reference to a string（引向一个字符串的引用），而 objectValue 是一个 const T：

```
template<class T>
class NamedObject {
public:
  // this ctor no longer takes a const name, because nameValue
  // is now a reference-to-non-const string. The char* constructor
  // is gone, because we must have a string to refer to.
  NamedObject(std::string& name, const T& value);

  ...                               // as above, assume no
                                    // operator= is declared

private:
  std::string& nameValue;           // this is now a reference
  const T objectValue;              // this is now const
};
```

现在，考虑这里会发生什么：

```
std::string newDog("Persephone");

std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);               // when I originally wrote this, our
                                             // dog Persephone was about to
                                             // have her second birthday
NamedObject<int> s(oldDog, 36);              // the family dog Satch (from my
                                             // childhood) would be 36 if she
                                             // were still alive

p = s;                                       // what should happen to
                                             // the data members in p?
```

assignment（赋值）之前，p.nameValue 和 s.nameValue 都引向 string objects（对象），虽然并非同一个。那个 assignment（赋值）对 p.nameValue 产生了什么影响呢？assignment（赋值）之后，p.nameValue 所引向的 string 是否就是 s.nameValue 所引向的那一个呢，也就是说，reference（引用）本身被改变了？如果是这样，就违反了常规，因为 C++ 并没有提供使一个 reference（引用）引向另一个 objects（对象）的方法。换一种思路，是不是 p.nameValue 所引向的那个 string objects（对象）被改变了，从而影响了其他 objects（对象）—— pointers（指针）或 references（引用）持续指向的那个 string，也就是，赋值中并没有直接涉及到的对象？这是 compiler-generated（编译器生成）的 copy assignment operator（拷贝赋值运算符）应该做的事情吗？

面对这个难题，C++ 拒绝编译代码。如果你希望一个包含 reference member（引用成员）的 class（类）支持 assignment（赋值），你必须自己定义 copy assignment operator（拷贝赋值运算符）。对于含有 const members（const 成员）的 classes（类），编译器会有类似的行为（就像上面那个改变后的 class（类）中的 objectValue）。改变 const members（const 成员）是不合法的，所以编译器隐式生成的 assignment function（赋值函数）无法确定该如何对待它们。最后，如果 base classes（基类）将 copy assignment operator（拷贝赋值运算符）声明为 private，编译器拒绝为从它继承的 derived classes（派生类）生成 implicit copy assignment operators（隐式拷贝赋值运算符）。毕竟，编译器为派生类生成的 copy assignment operator（拷贝赋值运算符）也要处理其 base class parts（基类构件）（参见 Item 12），但如果这样做，它们当然无法调用那些 derived classes（派生类）无权调用的 member functions（成员函数）。

Things to Remember

- 编译器可以隐式生成一个 class（类）的 default constructor（缺省构造函数），copy constructor（拷贝构造函数），copy assignment operator（拷贝赋值运算符）和 destructor（析构函数）。

Item 6: 如果你不想使用 compiler-generated functions（编译器生成函数），就明确拒绝

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

房地产代理商出售房屋，服务于这样的代理商的软件系统自然要有一个 class（类）来表示被出售的房屋：

```
class HomeForSale { ... };
```

每一个房地产代理商都会很快指出，每一件房产都是独特的——没有两件是完全一样的。在这种情况下，为 HomeForSale object（对象）做一个 copy（拷贝）的想法就令人不解了。你怎么能拷贝一个独一无二的东西呢？因此最好让类似这种企图拷贝 HomeForSale object（对象）的行为不能通过编译：

```
HomeForSale h1;

HomeForSale h2;

HomeForSale h3(h1);              // attempt to copy h1 — should
                                 // not compile!

h1 = h2;                         // attempt to copy h2 — should
                                 // not compile!
```

唉，防止这种编译的方法并非那么简单易懂。通常，如果你不希望一个 class（类）支持某种功能，你可以简单地不声明赋予它这种功能的函数。这个策略对于 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）不起作用，因为，就象 Item 5 中指出的，如果你不声明它们，而有人又想调用它们，编译器就会替你声明它们。

这就限制了你。如果你不声明 copy constructor（拷贝构造函数）或 copy assignment operator（拷贝赋值运算符），编译器也可以替你生成它们。你的 class（类）还是会支持 copying（拷贝）。另一方面，如果你声明了这些函数，你的 class（类）依然会支持 copying（拷贝）。而我们此时的目的却是 prevent copying（防止拷贝）！

解决这个问题的关键是所有的编译器生成的函数都是 public（公有）的。为了防止生成这些函数，你必须自己声明它们，但是你没有理由把它们声明为 public（公有）的。相反，应该将 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）声明为 private（私有）的。通过显式声明一个 member function（成员函数），可以防止编译器生成它自己的版本，而且将这个函数声明为 private（私有）的，可以防止别人调用它。

通常，这个方案并不十分保险，因为 member（成员）和 friend functions（友元函数）还是能够调用你的 private 函数。换句话说，除非你十分聪明地不 define（定义）它们。那么，当有人不小心地调用了它们，在 link-time（连接时）会出现错误。这个窍门——声明 member functions（成员函数）为 private 却故意不去实现它——确实很好，在 C++ 的 iostreams 库里，就有几个类用此方法 prevent copying（防止拷贝）。比如，看一下你用的标准库的实现中 ios_base，basic_ios 和 sentry 的 definitions（定义），你就会看到 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）被声明为 private 而且没有被定义的情况。

将这个窍门用到 HomeForSale 上，很简单：

```
class HomeForSale {
public:
  ...

private:
  ...
  HomeForSale(const HomeForSale&);            // declarations only
  HomeForSale& operator=(const HomeForSale&);
};
```

你会注意到，我省略了 functions' parameters（函数参数）的名字。这不是必须的，只是一个普通的惯例。毕竟，函数不会被实现，更少会被用到，有什么必要指定参数名呢？

对于上面的 class definition（类定义），编译器将阻止客户拷贝 HomeForSale objects（对象）的企图，如果你不小心在 member（成员）或 friend function（友元函数）中这样做了，连接程序会提出抗议。

将 link-time error（连接时错误）提前到编译时间也是可行的（早发现错误毕竟比晚发现好），通过不在 HomeForSale 本身中声明 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）为 private，而是在一个为 prevent copying（防止拷贝）而特意设计的 base class（基类）中声明。这个 base class（基类）本身非常简单：

```
class Uncopyable {
protected:                                   // allow construction
  Uncopyable() {}                            // and destruction of
  ~Uncopyable() {}                           // derived objects...

private:
  Uncopyable(const Uncopyable&);             // ...but prevent copying
  Uncopyable& operator=(const Uncopyable&);
};
```

为了阻止拷贝 HomeForSale objects（对象），我们现在必须让它从 Uncopyable 继承：

```
class HomeForSale: private Uncopyable {      // class no longer
  ...                                        // declares copy ctor or
};                                           // copy assign. operator
```

这样做是因为，如果有人——甚至是 member（成员）或 friend function（友元函数）——试图拷贝一个 HomeForSale objects（对象），编译器将试图生成一个 copy constructor（拷贝构造函数）和一个 copy assignment operator（拷贝赋值运算符）。就象 Item 12 解释的，这些函数的 compiler-generated versions（编译器生成版）会试图调用 base class（基类）的相应函数，而这些调用将被拒绝，因为在 base class（基类）中，拷贝操作是 private（私有）的。

Uncopyable 的实现和使用包含一些微妙之处，比如，从 Uncopyable 继承不必是 public（公有）的（参见 Item 32 和 39），而且 Uncopyable 的 destructor（析构函数）不必是 virtual（虚拟）的（参见 Item 7）。因为 Uncopyable 不包含数据，所以它符合 Item 39 描述的 empty base class optimization（空基类优化）的条件，但因为它是 base class（基类），此项技术的应用不能引入 multiple inheritance（多继承）（参见 Item 40）。反过来说，multiple inheritance（多继承）有时会使 empty base class optimization（空基类优化）失效（还是参见 Item 39）。通常，你可以忽略这些微妙之处，而且仅仅像此处演示的这样来使用 Uncopyable，因为它的工作就像在做广告。你还可以使用在 Boost（参见 Item 55）中的一个可用版本。那个 class（类）名为 noncopyable。那是一个好东西，我只是发现那个名字有点儿 un-（不……）嗯…… nonnatural（非自然）。

Things to Remember

- 为了拒绝编译器自动提供的机能，将相应的 member functions（成员函数）声明为 private，而且不要给出 implementations（实现）。使用一个类似 Uncopyable 的 base class（基类）是方法之一。

Item 7: 在 polymorphic base classes（多态基类）中将 destructors（析构函数）声明为 virtual（虚拟）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

有很多方法取得时间，所以有必要建立一个 TimeKeeper base class（基类），并为不同的计时方法建立 derived classes（派生类）：

```
class TimeKeeper {
public:
  TimeKeeper();
  ~TimeKeeper();
  ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

很多客户只是想简单地取得时间而不关心如何计算的细节，所以一个 factory function（工厂函数）——返回 a base class pointer to a newly-created derived class object（一个指向新建派生类对象的基类指针）的函数——可以被用来返回一个指向 timekeeping object（计时对象）的指针：

```
TimeKeeper* getTimeKeeper();       // returns a pointer to a dynamic-
                                   // ally allocated object of a class
                                   // derived from TimeKeeper
```

与 factory function（工厂函数）的惯例一致，getTimeKeeper 返回的 objects（对象）是建立在 heap（堆）上的，所以为了避免泄漏内存和其它资源，每一个返回的 objects（对象）被完全 deleted 是很重要的。

```
TimeKeeper *ptk = getTimeKeeper();  // get dynamically allocated object
                                    // from TimeKeeper hierarchy

...                                 // use it

delete ptk;                         // release it to avoid resource leak
```

Item 13 解释了为什么依赖客户执行删除任务是 error-prone（错误倾向），Item 18 解释了 factory function（工厂函数）的 interface（接口）应该如何改变以防止普通的客户错误，但这些在这里都是次要的，因为在这个 Item 中，我们将精力集中于上面的代码中一个更基本的缺陷：即使客户做对了每一件事，也无法预知程序将如何运转。

问题在于 getTimeKeeper 返回一个 pointer to a derived class object（指向派生类对象的指针）（比如 AtomicClock），那个 object（对象）经由一个 base class pointer（基类指针）（也就是一个 TimeKeeper* pointer）被删除，而且这个 base class（基类）(TimeKeeper) 有一个 non-virtual destructor（非虚拟析构函数）。祸端就在这里，因为 C++ 规定：当一个 derived class object（派生类对象）通过使用一个 pointer to a base class with a non-virtual destructor（指向带有非虚拟析构函数的基类的指针）被删除，则结果是未定义的。运行时比较典型的后果是 derived part of the object（这个对象的派生部分）不会被析构。如果 getTimeKeeper 返回一个指向 AtomicClock object（对象）的指针，则 object（对象）的 AtomicClock 部分（也就是在 AtomicClock class 中声明的 data members（数据成员））很可能不会被析构，AtomicClock 的 destructor（析构函数）也不会运行。然而，base class part（基类部分）（也就是 TimeKeeper 部分）很可能已被析构，这就导致了一个古怪的 "partially destroyed" object（“部分被析构”对象）。这是一个导致泄漏资源，破坏数据结构以及消耗大量调试时间的绝妙方法。

消除这个问题很简单：给 base class（基类）一个 virtual destructor（虚拟析构函数）。于是，删除一个 derived class object（派生类对象）的时候就有了你所期望的行为。将析构 entire object（整个对象），包括全部的 derived class parts（派生类构件）：

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  ...
};

TimeKeeper *ptk = getTimeKeeper();

...

delete ptk;                          // now behaves correctly
```

类似 TimeKeeper 的 base classes（基类）一般都包含除了 destructor（析构函数）以外的其它 virtual functions（虚拟函数），因为 virtual functions（虚拟函数）的目的就是允许 derived class implementations（派生类实现）的定制化（参见 Item 34）。例如，TimeKeeper 可以有一个 virtual functions（虚拟函数）getCurrentTime，它在各种不同的 derived classes（派生类）中有不同的实现。几乎所有拥有 virtual functions（虚拟函数）的 class（类）差不多都应该有一个 virtual destructor（虚拟析构函数）。

如果一个 class（类）不包含 virtual functions（虚拟函数），这经常预示不打算将它作为 base class（基类）使用。当一个 class（类）不打算作为 base class（基类）时，将 destructor（析构函数）虚拟通常是个坏主意。考虑一个表现二维空间中的点的

class（类）：

```
class Point {                           // a 2D point
public:
  Point(int xCoord, int yCoord);
  ~Point();

private:
  int x, y;
};
```

如果一个 int 占用 32 bits，一个 Point object 正好适用于 64-bit 的寄存器。而且，这样一个 Point object 可以被作为一个 64-bit 的量传递给其它语言写的函数，比如 C 或者 FORTRAN。如果 Point 的 destructor（析构函数）被虚拟，情况就完全不一样了。

virtual functions（虚拟函数）的实现要求 object（对象）携带额外的信息，这些信息用于在运行时确定该 object（对象）应该调用哪一个 virtual functions（虚拟函数）。典型情况下，这一信息具有一种被称为 vptr ("virtual table pointer")（虚拟函数表指针）的指针的形式。vptr 指向一个被称为 vtbl ("virtual table")（虚拟函数表）的 array of function pointers（函数指针数组），每一个带有 virtual functions（虚拟函数）的 class（类）都有一个相关联的 vtbl。当在一个 object（对象）上调用 virtual functions（虚拟函数）时，实际的被调用函数通过下面的步骤确定：找到 object（对象）的 vptr 指向的 vtbl，然后在 vtbl 中寻找合适的 function pointer（函数指针）。

virtual functions（虚拟函数）是如何实现的细节并不重要。重要的是如果 Point class 包含一个 virtual functions（虚拟函数），这个类型的 object（对象）的大小就会增加。在一个 32-bit 架构中，它们将从 64 bits（相当于两个 ints）长到 96 bits（两个 ints 加上 vptr）；在一个 64-bit 架构中，它们可能从 64 bits 长到 128 bits，因为在这样的架构中指针的大小是 64 bits 的。为 Point 加上 vptr 将会使它的大小增长 50-100%！Point object（对象）不再适合 64-bit 寄存器。而且，Point object（对象）在 C++ 和其它语言（比如 C）中，看起来不再具有相同的结构，因为它们在其它语言中的对应物没有 vptr。结果，Points 不再可能传入其它语言写成的函数或从其中传出，除非你为 vptr 做出明确的补偿，而这是它自己的实现细节并因此失去可移植性。

最终结果就是无故地将所有 destructors（析构函数）声明为 virtual（虚拟），和从不把它们声明为 virtual（虚拟）一样是错误的。实际上，很多人总结过这条规则：declare a virtual destructor in a class if and only if that class contains at least one virtual function（当且仅当一个类中包含至少一个虚拟函数时，则在类中声明一个虚拟析构函数）。

甚至在完全没有 virtual functions（虚拟函数）时，也有可能纠缠于 non-virtual destructor（非虚拟析构函数）问题。例如，标准 string 类型不包含 virtual functions（虚拟函数），但是被误导的程序员有时将它当作 base class（基类）使用：

```
class SpecialString: public std::string {   // bad idea! std::string has a
  ...                                       // non-virtual destructor
};
```

一眼看上去，这可能无伤大雅，但是，如果在程序的某个地方因为某种原因，你将一个 pointer-to-SpecialString（指向 SpecialString 的指针）转型为一个 pointer-to-string（指向 string 的指针），然后你将 delete 施加于这个 string pointer（指针），你就立刻被放逐到 undefined behavior（未定义行为）的领地：

```
SpecialString *pss =   new SpecialString("Impending Doom");

std::string *ps;
...
ps = pss;                           // SpecialString* → std::string*
...
delete ps;                          // undefined! In practice,
                                    // *ps's SpecialString resources
                                    // will be leaked, because the
                                    // SpecialString destructor won't
                                    // be called.
```

同样的分析可以适用于任何缺少 virtual destructor（虚拟析构函数）的 class（类），包括全部的 STL container（容器）类型（例如，vector，list，set，tr1::unordered_map（参见 Item 54）等）。如果你受到从 standard container（标准容器）或任何其它带有 non-virtual destructor（非虚拟析构函数）的 class（类）继承的诱惑，一定要挺住！（不幸的是，C++ 不提供类似 Java 的 final classes（类）或 C# 的 sealed classes（类）的 derivation-prevention mechanism（防派生机制）。）

有时候，给一个 class（类）提供一个 pure virtual destructor（纯虚拟析构函数）能提供一些便利。回想一下，pure virtual functions（纯虚拟函数）导致 abstract classes（抽象类）——不能被实例化的 classes（类）（也就是说你不能创建这个类型的 objects（对象））。然而，有时候，你有一个 class（类），你希望它是抽象的，但没有任何 pure virtual functions（纯虚拟函数）。怎么办呢？因为一个 abstract classes（抽象类）注定要被用作 base class（基类），又因为一个 base class（基类）应该有一个 virtual destructor（虚拟析构函数），还因为一个 pure virtual functions（纯虚拟函数）产生一个 abstract classes（抽象类），好了，解决方案很简单：在你想要变成抽象的 class（类）中声明一个 pure virtual destructor（纯虚拟析构函数）。这是一个例子：

```
class AWOV {                        // AWOV = "Abstract w/o Virtuals"
public:
  virtual ~AWOV() = 0;              // declare pure virtual destructor
};
```

这个 class（类）有一个 pure virtual functions（纯虚拟函数），所以它是抽象的，又因为它有一个 virtual destructor（虚拟析构函数），所以你不必担心析构函数问题。这是一个螺旋。然而，你必须为 pure virtual destructor（纯虚拟析构函数）提供一个 definition（定义）：

```
AWOV::~AWOV() {} // definition of pure virtual dtor
```

destructors（析构函数）的工作方式是：most derived class（层次最低的派生类）的 destructor（析构函数）最先被调用，然后调用每一个 base class（基类）的 destructors（析构函数）。编译器会生成一个从它的 derived classes（派生类）的 destructors（析构函数）对 ~AWOV 的调用，所以你不得不确保为函数提供一个函数体。如果你不这样做，连接程序会提出抗议。

为 base classes（基类）提供 virtual destructor（虚拟析构函数）的规则仅仅适用于 polymorphic base classes（多态基类）—— base classes（基类）被设计成允许通过 base class interfaces（基类接口）对 derived class types（派生类类型）进行操作。TimeKeeper 就是一个 polymorphic base classes（多态基类），因为即使我们只有类型为 TimeKeeper 的 pointers（指针）指向它们的时候，我们也期望能够操作 AtomicClock 和 WaterClock objects（对象）。

并非所有的 base classes（基类）都被设计用于 polymorphically（多态）。例如，无论是 standard string type（标准 string 类型），还是 STL container types（STL 容器类型）全被设计成 base classes（基类），可没有哪个是 polymorphic（多态）的。一些 classes（类）虽然被设计用于 base classes（基类），但并非被设计用于 polymorphically（多态）。这样的 classes（类）——例如 Item 6 中的 Uncopyable 和标准库中的 input_iterator_tag（参见 Item 47）——没有被设计成允许经由 base class interfaces（基类接口）对 derived class objects（派生类对象）进行操作。所以它们就不需要 virtual destructor（虚拟析构函数）。

Things to Remember

- polymorphic base classes（多态基类）应该声明 virtual destructor（虚拟析构函数）。如果一个 class（类）有任何 virtual functions（虚拟函数），它就应该有一个 virtual destructor（虚拟析构函数）。
- 不是设计用来作为 base classes（基类）或不是设计用于 polymorphically（多态）的 classes（类）就不应该声明 virtual destructor（虚拟析构函数）。

Item 8: 防止因为 exceptions（异常）而离开 destructors（析构函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 并不禁止从 destructors（析构函数）中引发 exceptions（异常），但是它坚决地阻止这样的实践。至于有什么好的理由，考虑：

```
class Widget {
public:
  ...
  ~Widget() { ... }              // assume this might emit an exception
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                // v is automatically destroyed here
```

当 vector v 被析构时，它有责任析构它包含的所有 Widgets。假设 v 中有十个 Widgets，在第一个的析构过程中，抛出一个 exception（异常）。其它 9 个 Widgets 仍然必须被析构（否则它们持有的所有资源将被泄漏），所以 v 应该调用它们的 destructors（析构函数）。但是假设在这个调用期间，第二个 Widget 的 destructors（析构函数）又抛出一个 exception（异常）。现在同时有两个活动的 exceptions（异常），对于 C++ 来说，这太多了。在非常巧合的条件下产生这样两个同时活动的异常，程序的执行会终止或者引发 undefined behavior（未定义行为）。在本例中，将引发 undefined behavior（未定义行为）。使用任何其它的标准库 container（容器）（比如，list，set），任何 TR1（参见 Item 54）中的 container（容器），甚至是一个 array（数组），都可能会引发同样的 undefined behavior（未定义行为）。也并非必须是 containers（容器）或 arrays（数组）才会陷入麻烦。程序过早终止或 undefined behavior（未定义行为）是 destructors（析构函数）引发 exceptions（异常）的结果，即使没有使用 containers（容器）和 arrays（数组）也会如此。C++ 不喜欢引发 exceptions（异常）的 destructors（析构函数）。

这比较容易理解，但是如果你的 destructor（析构函数）需要执行一个可能失败而抛出一个 exception（异常）的操作，该怎么办呢？例如，假设你与一个数据库连接类一起工作：

```
class DBConnection {
public:
  ...

  static DBConnection create();        // function to return
                                       // DBConnection objects; params
                                       // omitted for simplicity

  void close();                        // close connection; throw an
};                                     // exception if closing fails
```

为了确保客户不会忘记在 DBconnection objects（对象）上调用 close，一个合理的主意是为 DBConnection 建立一个 resource-managing class（资源管理类），在它的 destructor（析构函数）中调用 close。这样的 resource-managing classes（资源管理类）将在 Chapter 3（第三章）中一探究竟，但在这里，只要认为这样一个 class（类）的 destructor（析构函数）看起来像这样就足够了：

```
class DBConn {                          // class to manage DBConnection
public:                                 // objects
  ...
  ~DBConn()                             // make sure database connections
  {                                     // are always closed
   db.close();
  }
private:
  DBConnection db;
};
```

它允许客户像这样编程：

```
{                                       // open a block
   DBConn dbc(DBConnection::create());  // create DBConnection object
                                        // and turn it over to a DBConn
                                        // object to manage
 ...                                    // use the DBConnection object
                                        // via the DBConn interface
}                                       // at end of block, the DBConn
                                        // object is destroyed, thus
                                        // automatically calling close on
                                        // the DBConnection object
```

只要能成功地调用 close 就可以了，但是如果这个调用导致一个 exception（异常），DBConn 的 destructor（析构函数）将传播那个 exception（异常），也就是说，它将离开 destructor（析构函数）。这就产生了问题，因为 destructor（析构函数）抛出了一个烫手的山芋。

有两个主要的方法避免这个麻烦。DBConn 的 destructor（析构函数）能：

- Terminate the program if close tHRows（如果 close 抛出异常就终止程序），一般是通过调用 abort：

```
DBConn::~DBConn()
{
 try { db.close(); }
 catch (...) {
   make log entry that the call to close failed;
   std::abort();
 }
}
```

如果在析构的过程遭遇到错误后程序不能继续运行，这就是一个合理的选择。它有一个好处是：如果允许从 destructor（析构函数）传播 exception（异常）可能会导致 undefined behavior（未定义行为），这样就能防止它发生。也就是说，调用 abort 就可以预先防止 undefined behavior（未定义行为）。

- Swallow the exception arising from the call to close（抑制这个对 close 的调用造成的异常）：

```
DBConn::~DBConn()
{
 try { db.close(); }
 catch (...) {
      make log entry that the call to close failed;
 }
}
```

通常，swallowing exceptions（抑制异常）是一个不好的主意，因为它会隐瞒重要的信息——something failed（某事失败了）！然而，有些时候，swallowing exceptions（抑制异常）比冒程序过早终止或 undefined behavior（未定义行为）的风险更可取。程序必须能够在遭遇到一个错误并忽略之后还能继续可靠地运行，这才能成为一个可行的选择。

这些方法都不太吸引人。它们的问题首先在于程序无法对引起 close 抛出 exception（异常）的条件做出回应。

一个更好的策略是设计 DBConn 的 interface（接口），以使它的客户有机会对可能会发生的问题做出回应。例如，DBConn 能够自己提供一个 close 函数，从而给客户一个机会去处理从那个操作中发生的 exception（异常）。它还能保持对它的 DBConnection 是否已被 closed 的跟踪，如果没有就在 destructor（析构函数）中自己关闭它。这样可以防止连接被泄漏。如果在 DBConnection（原文如此，严重怀疑此处应为 DBConn ——译者注）的 destructor（析构函数）中对 close 的调用失败，无论如何，我们还可以再返回到终止或者抑制。

```
class DBConn {
public:
  ...

  void close()                                     // new function for
  {                                                // client use
    db.close();
    closed = true;
  }

  ~DBConn()
  {
   if (!closed) {
   try {                                            // close the connection
     db.close();                                    // if the client didn't
   }
   catch (...) {                                    // if closing fails,
     make log entry that call to close failed;      // note that and
     ...                                            // terminate or swallow
   }
  }

private:
  DBConnection db;
  bool closed;
};
```

将调用 close 的责任从 DBConn 的 destructor（析构函数）移交给 DBConn 的客户（同时在 DBConn 的 destructor（析构函数）中包含一个“候补”调用）可能会作为一种肆无忌惮地推卸责任的做法而使你吃惊。你甚至可以把它看作对 Item 18 中关于使 interfaces（接口）易于正确使用的建议的违背。实际上，这都不正确。如果一个操作可能失败而抛出一个 exception（异常），而且可能有必要处理这个 exception（异常），这个 exception（异常）就 has to come from some non-destructor function（必须来自非析构函数）。这是因为 destructor（析构函数）引发 exception（异常）是危险的，永远都要冒着程序过早终止或 undefined behavior（未定义行为）的风险。在本例中，让客户自己调用 close 并不是强加给他们的负担，而是给他们一个时机去应付错误，否则他们将没有机会做出回应。如果他们找不到可用到机会（或许因为他们相信不会有错误真的发生），他们可以忽略它，依靠 DBConn 的 destructor（析构函数）为他们调用 close。如果一个错误恰恰在那时发生——如果由 close 抛出——如果 DBConn 抑制了那个 exception（异常）或者终止了程序，他们将无处诉苦。毕竟，他们无处着手处理问题，他们将不再使用它。

Things to Remember

- destructor（析构函数）应该永不引发 exceptions（异常）。如果 destructor（析构函数）调用了可能抛出异常的函数，destructor（析构函数）应该捕捉所有异常，然后抑制它们或者终止程序。
- 如果 class（类）客户需要能对一个操作抛出的 exceptions（异常）做出回应，则那个 class（类）应该提供一个常规的函数（也就是说，non-destructor（非析构函数））来完成这个操作。

Item 9: 绝不要在 construction（构造）或 destruction（析构）期间调用 virtual functions（虚拟函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我以这个概述开始：你不应该在 construction（构造）或 destruction（析构）期间调用 virtual functions（虚拟函数），因为这样的调用不会如你想象那样工作，而且它们做的事情保证会让你很郁闷。如果你转为 Java 或 C# 程序员，也请你密切关注本 Item，因为在 C++ 急转弯的地方，那些语言也紧急转了一个弯。

假设你有一套模拟股票交易的 class hierarchy（类继承体系），例如，购入订单，出售订单等。对于这样的交易来说可供审查是非常重要的，所每次一个交易对象被创建，在一个审查日志中就需要创建一个相应的条目。下面是一个看起来似乎合理的解决问题的方法：

```
class Transaction {                               // base class for all
public:                                           // transactions
  Transaction();

  virtual void logTransaction() const = 0;        // make type-dependent
                                                  // log entry
  ...
};

Transaction::Transaction()                        // implementation of
{                                                 // base class ctor
  ...
  logTransaction();                               // as final action, log this
}                                                 // transaction

class BuyTransaction: public Transaction {        // derived class
public:
  virtual void logTransaction() const;            // how to log trans-
                                                  // actions of this type
  ...
};

class SellTransaction: public Transaction {       // derived class
public:
 virtual void logTransaction() const;             // how to log trans-
                                                  // actions of this type
  ...
};
```

考虑执行这行代码时会发生什么：

```
BuyTransaction b;
```

很明显一个 BuyTransaction 的 constructor（构造函数）会被调用，但是首先，一个 Transaction 的 constructor（构造函数）必须先被调用，derived class objects（派生类对象）中的 base class parts（基类构件）先于 derived class parts（派生类构件）被构造。Transaction 的 constructor（构造函数）的最后一行调用 virtual functions（虚拟函数）logTransaction，但是结果会让你大吃一惊，被调用的 logTransaction 版本是在 Transaction 中的那一个，而不是 BuyTransaction 中的那一个——即使被创建的 object （对象）类型是 BuyTransaction。base class construction（基类构造）期间，virtual functions（虚拟函数）从来不会 go down（向下匹配）到 derived classes（派生类）。取而代之的是，那个 object （对象）的行为好像它就是 base type（基类型）。非正式地讲，base class construction（基类构造）期间，virtual functions（虚拟函数）被禁止。

这个表面上看起来匪夷所思的行为存在一个很好的理由。因为 base class constructors（基类构造函数）在 derived class constructors（派生类构造函数）之前执行，当 base class constructors（基类构造函数）运行时，derived class data members（派生类数据成员）还没有被初始化。如果 base class construction（基类构造）期间 virtual functions（虚拟函数）的调用 went down（向下匹配）到 derived classes（派生类），derived classes（派生类）的函数差不多总会涉及到 local data members（局部数据成员），但是那些 data members（数据成员）至此还没有被初始化。这就会为 undefined behavior（未定义行为）和通宵达旦的调试噩梦开了一张通行证。调用涉及到一个 object（对象）还没有被初始化的构件自然是危险的，所以 C++ 告诉你此路不通。

实际上还有比这更基本的原理。在一个 derived class object（派生类对象）的 base class construction（基类构造）期间，object（对象）的类型是 base class（基类）的类型。不仅 virtual functions（虚拟函数）会解析到 base class（基类），而且用到 runtime type information（运行时类型信息）的语言构件（例如，dynamic_cast（参见 Item 27）和 typeid），也会将那个 object（对象）视为 base class type（基类类型）。在我们的例子中，当 Transaction 的 constructor（构造函数）运行到初始化一个 BuyTransaction object（对象）的 base class（基类）部分时，那个 object（对象）的是 Transaction 类型。C++ 的每一个构件将以如下眼光来看待它，而且这种看法是合理的：这个 object（对象）的 BuyTransaction-specific 的构件还没有被初始化，所以对它们视若无睹是最安全的。直到 derived class constructor（派生类构造函数）的执行开始之前，一个 object（对象）不会成为一个 derived class object（派生类对象）。

同样的推理也适用于 destruction（析构）。一旦 derived class destructor（派生类析构函数）运行，这个 object（对象）的 derived class data members（派生类数据成员）就呈现为未定义的值，所以 C++ 就将它们视为不再存在。在进入 base class destructor（基类析构函数）时，这个 object（对象）就成为一个 base class object（基类对象），C++ 的所有构件—— virtual functions（虚拟函数），dynamic_casts 等——都以此看待它。

在上面的示例代码中，Transaction 的 constructor（构造函数）造成了对一个 virtual functions（虚拟函数）的一次直接调用，是对本 Item 的指导建议的显而易见的违背。这一违背是如此显见，以致一些编译器会给出一个关于它的警告。（另一些则不会。参见 Item 53 对

于警告的讨论。）即使没有这样的一个警告，这个问题也几乎肯定会在运行之前暴露出来，因为 logTransaction 函数在 Transaction 中是 pure virtual（纯虚拟）的。除非它被定义（不太可能，但确实可能——参见 Item 34），否则程序将无法连接：连接程序无法找到 Transaction::logTransaction 的必要的实现。

在 construction（构造）或 destruction（析构）期间调用 virtual functions（虚拟函数）的问题并不总是如此容易被察觉。如果 Transaction 有多个 constructors（构造函数），每一个都必须完成一些相同的工作，软件工程为避免代码重复，将共通的 initialization（初始化）代码，包括对 logTransaction 的调用，放入一个 private non-virtual initialization function（私有非虚拟初始化函数）中，叫做 init：

```
class Transaction {
public:
  Transaction()
  { init(); }                                    // call to non-virtual...

  virtual void logTransaction() const = 0;
  ...

private:
  void init()
  {
    ...
    logTransaction();                            // ...that calls a virtual!
  }
};
```

这个代码在概念上和早先那个版本相同，但是它更阴险，因为一般来说它会躲过编译器和连接程序的抱怨。在这种情况下，因为 logTransaction 在 Transaction 中是 pure virtual（纯虚的），在 pure virtual（纯虚）被调用时，大多数 runtime systems（运行时系统）会异常中止那个程序（一般会对此结果给出一条消息）。然而，如果 logTransaction 在 Transaction 中是一个 "normal" virtual function（“常规”虚拟函数）（也就是说，not pure virtual（非纯虚拟的）），而且带有一个实现，那个版本将被调用，程序会继续一路小跑，让你想象不出为什么在 derived class object（派生类对象）被创建的时候会调用 logTransaction 的错误版本。避免这个问题的唯一办法就是确保你的 constructors（构造函数）或 destructors（析构函数）决不在被创建或析构的 object（对象）上调用 virtual functions（虚拟函数），它们所调用的全部函数也要服从同样的约束。

但是，你如何确保在每一次 Transaction hierarchy（继承体系）中的一个 object（对象）被创建时，都会调用 logTransaction 的正确版本呢？显然，在 Transaction constructor(s)（构造函数）中在这个 object（对象）上调用 virtual functions（虚拟函数）的做法是错误的。

有不同的方法来解决这个问题。其中之一是将 Transaction 中的 logTransaction 转变为一个 non-virtual function（非虚拟函数），这就需要 derived class constructors（派生类构造函数）将必要的日志信息传递给 Transaction constructor（构造函数）。那个函数就可以安全地调用 non-virtual（非虚拟）的 logTransaction。如下：

```
class Transaction {
public:
  explicit Transaction(const std::string& logInfo);

  void logTransaction(const std::string& logInfo) const;   // now a non-
                                                            // virtual func
  ...
};

Transaction::Transaction(const std::string& logInfo)
{
  ...
  logTransaction(logInfo);                                  // now a non-
}                                                           // virtual call

class BuyTransaction: public Transaction {
public:
 BuyTransaction( parameters )
 : Transaction(createLogString( parameters ))               // pass log info
  { ... }                                                   // to base class
   ...                                                      // constructor

private:
  static std::string createLogString( parameters );
};
```

换句话说，由于你不能在 base classes（基类）的 construction（构造）过程中使用 virtual functions（虚拟函数）向下匹配，你可以改为让 derived classes（派生类）将必要的构造信息上传给 base class constructors（基类构造函数）作为补偿。

在此例中，注意 BuyTransaction 中那个 (private) static 函数 createLogString 的使用。使用一个辅助函数创建一个值传递给 base class constructors（基类构造函数），通常比通过在 member initialization list（成员初始化列表）给 base class（基类）它所需要的东西更加便利（也更加具有可读性）。将那个函数做成 static，就不会有偶然触及到一个新生的 BuyTransaction object（对象）的 as-yet-uninitialized data members（仍未初始化的数据成员）的危险。这很重要，因为实际上那些 data members（数据成员）处在一个未定义状态，这就是为什么在 base class（基类）construction（构造）和 destruction（析构）期间调用 virtual functions（虚拟函数）不能首先向下匹配到 derived classes（派生类）的原因。

Things to Remember

- 在 construction（构造）或 destruction（析构）期间不要调用 virtual functions（虚拟函数），因为这样的调用不会转到比当前执行的 constructor（构造函数）或 destructor（析构函数）所属的 class（类）更深层的 derived class（派生类）。

Item 10: 让 assignment operators（赋值运算符）返回一个 reference to *this（引向 *this 的引用）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

关于 assignments（赋值）的一件有意思的事情是你可以把它们穿成一串。

```
int x, y, z;

x = y = z = 15; // chain of assignments
```

另一件有意思的事情是 assignment（赋值）是 right-associative（右结合）的，所以，上面的赋值串可以解析成这样：

```
x = (y = (z = 15));
```

这里，15 赋给 z，然后将这个 assignment（赋值）的结果（最新的 z）赋给 y，然后将这个 assignment（赋值）的结果（最新的 y）赋给 x。

这里的实现方法是让 assignment（赋值）返回一个 reference to its left-hand argument（引向它的左侧参数的引用），而且这就是当你为你的 classes（类）实现 assignment operators（赋值运算符）时应该遵守的惯例：

```
class Widget {
public:
  ...
Widget& operator=(const Widget& rhs)   // return type is a reference to
{                                      // the current class
  ...
  return *this;                        // return the left-hand object
  }
  ...
};
```

这个惯例适用于所有的 assignment operators（赋值运算符），而不仅仅是上面那样的标准形式。因此：

```
class Widget {
public:
  ...
  Widget& operator+=(const Widget& rhs   // the convention applies to
  {                                      // +=, -=, *=, etc.
   ...
   return *this;
  }
   Widget& operator=(int rhs)            // it applies even if the
   {                                     // operator's parameter type
      ...                                // is unconventional
      return *this;
   }
   ...
};
```

这仅仅是一个惯例，代码并不会按照这个意愿编译。然而，这个惯例被所有的 built-in types（内建类型）和标准库中（或者即将进入标准库——参见 Item 54）的 types（类型）（例如，string，vector，complex，tr1::shared_ptr 等）所遵守。除非你有好的做不同事情理由，否则，不要破坏它。

Things to Remember

- 让 assignment operators（赋值运算符）返回一个 reference to *this（引向 *this 的引用）。

Item 11: 在 operator= 中处理 assignment to self（自赋值）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

当一个 object（对象）赋值给自己的时候就发生了一次 assignment to self（自赋值）：

```
class Widget { ... };

Widget w;
...

w = w;                                   // assignment to self
```

这看起来很愚蠢，但它是合法的，所以应该确信客户会这样做。另外，assignment（赋值）也并不总是那么容易辨别。例如，

```
a = a[j]; // potential assignment to self
```

如果 i 和 j 有同样的值就是一个 assignment to self（自赋值），还有

```
*px = *py; // potential assignment to self
```

如果 px 和 py 碰巧指向同一个东西，这也是一个 assignment to self（自赋值）。这些不太明显的 assignments to self（自赋值）是由 aliasing（别名）（有不止一个方法引用一个 object（对象））造成的。通常，使用 references（引用）或者 pointers（指针）操作相同类型的多个 objects（对象）的代码需要考虑那些 objects（对象）可能相同的情况。实际上，如果两个 objects（对象）来自同一个 hierarchy（继承体系），甚至不需要公开声明，它们就是相同类型的，因为一个 base class（基类）的 reference（引用）或者 pointer（指针）也能够引向或者指向一个 derived class（派生类）类型的 object（对象）：

```
class Base { ... };

class Derived: public Base { ... };

void doSomething(const Base& rb,                    // rb and *pd might actually be
                 Derived* pd);                      // the same object
```

如果你遵循 Item 13 和 14 的建议，你应该总是使用 objects（对象）来管理 resources（资源），而且你应该确保那些 resource-managing objects（资源管理对象）被拷贝时行为良好。在这种情况下，你的 assignment operators（赋值运算符）在你没有考虑自赋值的时候可能也是 self-assignment-safe（自赋值安全）的。然而，如果你试图自己管理 resources（资源）（如果你正在写一个 resource-managing class（资源管理类），你当然必须这样做），你可能会落入在你用完一个 resource（资源）之前就已意外地将它释放的陷阱。例如，假设你创建了一个 class（类），它持有一个指向动态分配 bitmap（位图）的 raw pointer（裸指针）：

```
class Bitmap { ... };

class Widget {
  ...

private:
  Bitmap *pb;                                     // ptr to a heap-allocated object
};
```

下面是一个表面上看似合理 operator= 的实现，但如果出现 assignment to self（自赋值）则是不安全的。（它也不是 exception-safe（异常安全）的，但我们要过一会儿才会涉及到它。）

```
Widget&
Widget::operator=(const Widget& rhs)              // unsafe impl. of operator=
{
  delete pb;                                      // stop using current bitmap
  pb = new Bitmap(*rhs.pb);                       // start using a copy of rhs's bitmap

  return *this;                                   // see Item 10
}
```

这里的 self-assignment（自赋值）问题在 operator= 的内部，*this（赋值的目标）和 rhs 可能是同一个 object（对象）。如果它们是，则那个 delete 不仅会销毁 current object（当前对象）的 bitmap（位图），也会销毁 rhs 的 bitmap（位图）。在函数的结尾，Widget——通过 assignment to self（自赋值）应该没有变化——发现自己持有一个指向已删除 object（对象）的指针。

防止这个错误的传统方法是在 operator= 的开始处通过 identity test（一致性检测）来阻止 assignment to self（自赋值）：

```
Widget& Widget::operator=(const Widget& rhs)
{
  if (this == &rhs) return *this;   // identity test: if a self-assignment,
                                    // do nothing
  delete pb;
  pb = new Bitmap(*rhs.pb);

  return *this;
}
```

这个也能工作，但是我在前面提及那个 operator= 的早先版本不仅仅是 self-assignment-unsafe（自赋值不安全）的，它也是 exception-unsafe（异常不安全）的，而且这个版本还有异常上的麻烦。详细地说，如果 "new Bitmap" 表达式引发一个 exception（异常）（可能因为供分配的内存不足或者因为 Bitmap 的 copy constructor（拷贝构造函数）抛出一个异常），Widget 将以持有一个指向被删除的 Bitmap 的指针而告终。这样的指针是有毒的，你不能安全地删除它们。你甚至不能安全地读取它们。你对它们唯一能做的安全的事情大概就是花费大量的调试精力来断定它们起因于哪里。

幸亏，使 operator= exception-safe（异常安全）一般也同时弥补了它的 self-assignment-safe（自赋值安全）。这就导致了更加通用的处理 self-assignment（自赋值）问题的方法就是忽略它，而将焦点集中于达到 exception safety（异常安全）。Item 29 更加深入地探讨了 exception safety（异常安全），但是在本 Item 中，已经足以看出，在很多情况下，仔细地调整一下语句的顺序就可以得到 exception-safe（异常安全）（同时也是 self-assignment-safe（自赋值安全））的代码。例如，在这里，我们只要注意不要删除 pb，直到我们拷贝了它所指向的目标之后：

```
Widget& Widget::operator=(const Widget& rhs)
{
  Bitmap *pOrig = pb;               // remember original pb
  pb = new Bitmap(*rhs.pb);         // make pb point to a copy of *pb
  delete pOrig;                     // delete the original pb

  return *this;
}
```

现在，如果 "new Bitmap" 抛出一个 exception（异常），pb（以及它所在的 Widget）的遗迹没有被改变。甚至不需要 identity test（一致性检测），这里的代码也能处理 assignment to self（自赋值），因为我们做了一个原始 bitmap（位图）的拷贝，删除原始 bitmap（位图），然后指向我们作成的拷贝。这可能不是处理 self-assignment（自赋值）的最有效率的做法，但它能够工作。

如果你关心效率，你可以在函数开始处恢复 identity test（一致性检测）。然而，在这样做之前，先问一下自己，你认为 self-assignments（自赋值）发生的频率是多少，因为这个检测不是免费午餐。它将使代码（源代码和目标代码）有少量增大，而且它将在控制流中引入一个分支，这两点都会降低运行速度。例如，instruction prefetching（指令预读），caching（缓存）和 pipelining（流水线操作）的效力都将被降低。

另一个可选的手动排列 operator= 中语句顺序以确保实现是 exception- and self-assignment-safe（异常和自赋值安全）的方法是使用被称为 "copy and swap" 的技术。这一技术和 exception safety（异常安全）关系密切，所以将在 Item 29 中描述。然而，这是一个写 operator= 的足够通用的方法，值得一看，这样一个实现看起来通常就像下面这样：

```
class Widget {
  ...
  void swap(Widget& rhs);       // exchange *this's and rhs's data;
  ...                           // see Item 29 for details
};

Widget& Widget::operator=(const Widget& rhs)
{
  Widget temp(rhs);             // make a copy of rhs's data

  swap(temp);                   // swap *this's data with the copy's
  return *this;
}
```

在这个主题上的一个变种利用了如下事实：（1）一个 clsaa（类）的 copy assignment（拷贝赋值运算符）可以被声明为 take its argument by value（以传值方式取得它的参数）；（2）通过传值方式传递某些东西以做出它的一个 copy（拷贝）（参见 Item 20）：

```
Widget& Widget::operator=(Widget rhs)   // rhs is a copy of the object
{                                       // passed in — note pass by val

  swap(rhs);                            // swap *this's data with
                                        // the copy's
  return *this;
}
```

对我个人来说，我担心这个方法在灵活的祭坛上牺牲了清晰度，但是通过将拷贝操作从函数体中转移到参数的构造中，有时能使编译器产生更有效率的代码倒也是事实。

Things to Remember

- 当一个 object（对象）被赋值给自己的时候，确保 operator= 是行为良好的。技巧包括比较 source（源）和 target objects（目标对象）的地址，关注语句顺序，和 copy-and-swap。
- 如果两个或更多 objects（对象）相同，确保任何操作多于一个 object（对象）的函数行为正确。

Item 12: 拷贝一个对象的所有组成部分

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在设计良好的面向对象系统中，封装了对象内部的配件，仅留两个函数用于对象的拷贝：一般称为拷贝构造函数（copy constructor）和拷贝赋值运算符（copy assignment operator）。我们将它们统称为拷贝函数（copying functions）。Item 5 讲述了如果需要，编译器会生成拷贝函数，而且阐明了编译器生成的版本正象你所期望的：它们拷贝被拷贝对象的全部数据。

当你声明了你自己的拷贝函数，你就是在告诉编译器你不喜欢缺省实现中的某些东西。编译器对此好像怒发冲冠，而且它们会用一种古怪的方式报复：当你的实现存在一些几乎可以确定错误时，它偏偏不告诉你。

考虑一个象征消费者（customers）的类，这里的拷贝函数是手写的，以便将对它们的调用记入日志：

```
void logCall(const std::string& funcName);        // make a log entry

class Customer {
public:
  ...
  Customer(const Customer& rhs);
  Customer& operator=(const Customer& rhs);
  ...

private:
  std::string name;
};
Customer::Customer(const Customer& rhs)
: name(rhs.name)                                  // copy rhs's data
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");

  name = rhs.name;                                // copy rhs's data

  return *this;                                   // see Item 10
}
```

这里的每一件事看起来都不错，实际上也确实不错——直到 Customer 中加入了另外的数据成员：

```
class Date { ... };       // for dates in time

class Customer {
public:
  ...                     // as before

private:
  std::string name;
  Date lastTransaction;
};
```

在这里，已有的拷贝函数只进行了部分拷贝：它们拷贝了 Customer 的 name，但没有拷贝它的 lastTransaction。然而，大部分编译器对此毫不在意，即使是在最高的警告级别（maximal warning level）（参见 Item 53）。这是它们在对你写自己的拷贝函数进行报复。你拒绝了它们写的拷贝函数，所以如果你的代码是不完善的，他们也不告诉你。结论显而易见：如果你为一个类增加了一个数据成员，你务必要做到更新拷贝函数。（你还需要更新类中的全部的构造函数（参见 Item 4 和 45）以及任何非标准形式的 operator=（Item 10 给出了一个例子）。如果你忘记了，编译器未必会提醒你。）

这个问题最为迷惑人的情形之一是它会通过继承发生。考虑：

```
class PriorityCustomer: public Customer {                  // a derived class
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...

private:
   int priority;
};
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
: priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  priority = rhs.priority;

  return *this;

}
```

PriorityCustomer 的拷贝函数看上去好像拷贝了 PriorityCustomer 中的每一样东西，但是再看一下。是的，它确实拷贝了 PriorityCustomer 声明的数据成员，但是每个 PriorityCustomer 还包括一份它从 Customer 继承来的数据成员的副本，而那些数据成员根本没有被拷贝！PriorityCustomer 的拷贝构造函数没有指定传递给它的基类构造函数的参数（也就是说，在它的成员初始化列表中没有提及 Customer），所以，PriorityCustomer 对象的 Customer 部分

被 Customer 的构造函数在无参数的情况下初始化——使用缺省构造函数。（假设它有，如果没有，代码将无法编译。）那个构造函数为 name 和 lastTransaction 进行一次缺省的初始化。

对于 PriorityCustomer 的拷贝赋值运算符，情况有些微的不同。它不会试图用任何方法改变它的基类的数据成员，所以它们将保持不变。

无论何时，你打算自己为一个派生类写拷贝函数时，你必须注意同时拷贝基类部分。那些地方的典型特征当然是 private（参见 Item 22），所以你不能直接访问它们。派生类的拷贝函数必须调用和它们对应的基类函数：

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
:    Customer(rhs),                   // invoke base class copy ctor
  priority(rhs.priority)
{
  logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
  logCall("PriorityCustomer copy assignment operator");

  Customer::operator=(rhs);           // assign base class parts
  priority = rhs.priority;

  return *this;
}
```

本 Item 标题中的 "copy all parts" 的含义现在应该清楚了。当你写一个拷贝函数，需要保证（1）拷贝所有本地数据成员以及（2）调用所有基类中的适当的拷贝函数。

在实际中，两个拷贝函数经常有相似的函数体，而这一点可能吸引你试图通过用一个函数调用另一个来避免代码重复。你希望避免代码重复的想法值得肯定，但是用一个拷贝函数调用另一个来做到这一点是错误的。

用拷贝赋值运算符调用拷贝构造函数是没有意义的，因为你这样做就是试图去构造一个已经存在的对象。这太荒谬了，甚至没有一种语法来支持它。有一种语法看起来好像能让你这样做，但实际上你做不到，还有一种语法采用迂回的方法这样做，但它们在某种条件下会对破坏你的对象。所以我不打算给你看任何那样的语法。无条件地接受这个观点：不要用拷贝赋值运算符调用拷贝构造函数。

尝试一下另一种相反的方法——用拷贝构造函数调用拷贝赋值运算符——这同样是荒谬的。一个构造函数初始化新的对象，而一个赋值运算符只能用于已经初始化过的对象。借助构造过程给一个对象赋值将意味着对一个尚未初始化的对象做一些事，而这些事只有用于已初始化对象才有意义。简直是胡搞！不要做这种尝试。

作为一种代替，如果你发现你的拷贝构造函数和拷贝赋值运算符有相似的代码，通过创建第三个供两者调用的成员函数来消除重复。这样的函数当然是 private 的，而且经常叫做 init。这一策略是在拷贝构造函数和拷贝赋值运算符中消除代码重复的安全的，被证实过的方法。

Things to Remember

- 拷贝函数应该保证拷贝一个对象的所有数据成员以及所有的基类部分。
- 不要试图依据一个拷贝函数实现另一个。作为代替，将通用功能放入第三个供双方调用的函数。

Item 13: 使用对象管理资源

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们和一个投资（例如，股票，债券等）模型库一起工作，各种各样的投资形式从一个根类 Investment 派生出来：

```
class Investment { ... };            // root class of hierarchy of
                                     // investment types
```

进一步假设这个库使用了通过一个 factory 函数（参见 Item 7）为我们提供特定 Investment 对象的方法：

```
Investment* createInvestment();      // return ptr to dynamically allocated
                                     // object in the Investment hierarchy;
                                     // the caller must delete it
                                     // (parameters omitted for simplicity)
```

通过注释指出，当 createInvestment 函数返回的对象不再使用时，由 createInvestment 的调用者负责删除它。那么，请考虑，写一个函数 f 来履行以下职责：

```
void f()
{
  Investment *pInv = createInvestment();         // call factory function
  ...                                            // use pInv
  delete pInv;                                   // release object
}
```

这个看上去没问题，但是有几种情形会造成 f 在删除它从 createInvestment 得到的 investment 对象时失败。有可能在这个函数的 "..." 部分的某处有一个提前出现的 return 语句。如果这样一个 return 执行了，控制流程就再也无法到达 delete 语句。还可能发生的一个类似情况是如果 createInvestment 的使用和删除在一个循环里，而这个循环以一个 continue 或 goto 语句提前退出。还有，"..." 中的一些语句可能抛出一个异常。如果这样，控制流程不会再到达那个 delete。无论那个 delete 被如何跳过，我们泄漏的不仅仅是容纳 investment 对象的内存，还包括那个对象持有的任何资源。

当然，小心谨慎地编程能防止这各种错误，但考虑到这些代码可能会随着时间的流逝而发生变化。为了对软件进行维护，一些人可能会在没有完全把握对这个函数的资源管理策略的其它部分的影响的情况下增加一个 return 或 continue 语句。尤有甚者，f 的 "..." 部分可能调用

了一个从不惯于抛出异常的函数，但是在它被“改良”后突然这样做了。依赖于 f 总能到达它的 delete 语句根本靠不住。

为了确保 createInvestment 返回的资源总能被释放，我们需要将那些资源放入一个类中，这个类的析构函数在控制流程离开 f 的时候会自动释放资源。实际上，这只是本 Item 介绍的观念的一半：将资源放到一个对象的内部，我们可以依赖 C++ 的自动地调用析构函数来确保资源被释放。（过一会儿我们还要介绍本 Item 观念的另一半。）

许多资源都是动态分配到堆上的，并在一个单独的块或函数内使用，而且应该在控制流程离开那个块或函数的时候释放。标准库的 auto_ptr 正是为这种情形量体裁衣的。auto_ptr 是一个类似指针的对象（一个智能指针），它的析构函数自动在它指向的东西上调用 delete。下面就是如何使用 auto_ptr 来预防 f 的潜在的资源泄漏：

```
void f()
{
  std::auto_ptr<Investment> pInv(createInvestment());  // call factory
                                                       // function
  ...                                                  // use pInv as
                                                       // before
}                                                      // automatically
                                                       // delete pInv via
                                                       // auto_ptr's dtor
```

这个简单的例子示范了使用对象管理资源的两个重要的方面：

- 获得资源后应该立即移交给资源管理对象。如上，createInvestment 返回的资源被用来初始化即将用来管理它的 auto_ptr。实际上，因为获取一个资源并在同一个语句中初始化资源管理对象是如此常见，所以使用对象管理资源的观念也常常被称为 Resource Acquisition Is Initialization (RAII)。有时被获取的资源是被赋值给资源管理对象的，而不是初始化它们，但这两种方法都是在获取资源的同时就立即将它移交给资源管理对象。

- 资源管理对象使用它们的析构函数确保资源被释放。因为当一个对象被销毁时（例如，当一个对象离开其活动范围）会自动调用析构函数，无论控制流程是怎样离开一个块的，资源都会被正确释放。如果释放资源的动作会引起异常抛出，事情就会变得棘手，不过，关于那些问题请访问 Item 8，所以我们不必担心它。

因为当一个 auto_ptr 被销毁的时候，会自动删除它所指向的东西，所以不要让超过一个的 auto_ptr 指向同一个对象非常重要。如果发生了这种事情，那个对象就会被删除超过一次，而且会让你的程序通过捷径进入未定义行为。为了防止这个问题，auto_ptrs 具有不同寻常的特性：拷贝它们（通过拷贝构造函数或者拷贝赋值运算符）就是将它们置为空，拷贝的指针被设想为资源的唯一所有权。

```
sstd::auto_ptr<Investment>                    // pInv1 points to the
  pInv1(createInvestment());                 // object returned from
                                             // createInvestment

std::auto_ptr<Investment> pInv2(pInv1);      // pInv2 now points to the
                                             // object; pInv1 is now null

pInv1 = pInv2;                               // now pInv1 points to the
                                             // object, and pInv2 is null
```

这个奇怪的拷贝行为，增加了潜在的需求，就是通过 auto_ptrs 管理的资源必须绝对没有超过一个 auto_ptr 指向它们，这也就意味着 auto_ptrs 不是管理所有动态分配资源的最好方法。例如，STL 容器要求其内含物能表现出“正常的”拷贝行为，所以 auto_ptrs 的容器是不被允许的。

相对于 auto_ptrs，另一个可选方案是一个引用计数智能指针（reference-counting smart pointer, RCSP）。一个 RCSP 是一个智能指针，它能持续跟踪有多少对象指向一个特定的资源，并能够在不再有任何东西指向那个资源的时候删除它。就这一点而论，RCSP 提供的行为类似于垃圾收集（garbage collection）。与垃圾收集不同的是，无论如何，RCSP 不能打破循环引用（例如，两个没有其它使用者的对象互相指向对方）。

TR1 的 tr1::shared_ptr（参见 Item 54）是一个 RCSP，所以你可以这样写 f：

```
void f()
{
  ...

  std::tr1::shared_ptr<Investment>
    pInv(createInvestment());                // call factory function
  ...                                        // use pInv as before
}                                            // automatically delete
                                             // pInv via shared_ptr's dtor
```

这里的代码看上去和使用 auto_ptr 的几乎相同，但是拷贝 shared_ptrs 的行为却自然得多：

```
void f()
{
  ...

  std::tr1::shared_ptr<Investment>            // pInv1 points to the
    pInv1(createInvestment());                // object returned from
                                              // createInvestment

  std::tr1::shared_ptr<Investment>            // both
pInv1 and pInv2 now
    pInv2(pInv1);                             // point to the object

  pInv1 = pInv2;                              // ditto – nothing has
                                              // changed
  ...
}                                             // pInv1 and pInv2 are
                                              // destroyed, and the
                                              // object they point to is
                                              // automatically deleted
```

因为拷贝 tr1::shared_ptrs 的工作“符合预期”，它们能被用于 STL 容器以及其它和 auto_ptr 的非正统的拷贝行为不相容的环境中。

不要搞错，本 Item 不是关于 auto_ptr，tr1::shared_ptr 或任何其它种类的智能指针。而是关于使用对象管理资源的重要性的。auto_ptr 和 tr1::shared_ptr 仅仅是做这些事的对象的例子。（关于 tr1::shared_ptr 的更多信息，请参考 Item 14，18 和 54。）

auto_ptr 和 tr1::shared_ptr 都在它们的析构函数中使用 delete，而不是 delete []。（Item 16 描述两者的差异。）这就意味着将 auto_ptr 或 tr1::shared_ptr 用于动态分配的数组是个馊主意，可是，可悲的是，那居然可以编译：

```
std::auto_ptr<std::string>                        // bad idea! the wrong
  aps(new std::string[10]);                       // delete form will be used

std::tr1::shared_ptr<int> spi(new int[1024]);    // same problem
```

你可能会吃惊地发现 C++ 中没有可用于动态分配数组的类似 auto_ptr 或 tr1::shared_ptr 这样的东西，甚至在 TR1 中也没有。那是因为 vector 和 string 几乎总是能代替动态分配数组。如果你依然觉得有可用于数组的类似 auto_ptr 和类似 tr1::shared_ptr 的类更好一些的话，可以去看看 Boost（参见 Item 55）。在那里，你将高兴地找到 boost::scoped_array 和 boost::shared_array 两个类提供你在寻找的行为。

本 Item 的关于使用对象管理资源的指导间接表明：如果你手动释放资源（例如，使用 delete，而不使用资源管理类），你就是在自找麻烦。像 auto_ptr 和 tr1::shared_ptr 这样的预制的资源管理类通常会使本 Item 的建议变得容易，但有时，你使用了一个资源，而这些预加工的类不能如你所愿地做事。如果碰上这种情况，你就需要精心打造你自己的资源管理类。那也并非困难得可怕，但它包含一些需要你细心考虑的微妙之处。那些需要考虑的事项是 Item 14 和 15 的主题。

作为最后的意见，我必须指出 createInvestment 的裸指针（raw pointer）的返回形式就是资源泄漏的请帖，因为调用者忘记在他们取回来的指针上调用 delete 实在是太容易了。（即使他们使用一个 auto_ptr 或 tr1::shared_ptr 来完成 delete，他们仍然必须记住将 createInvestment 的返回值存储到智能指针对象中。）对付这个问题需要改变 createInvestment 的接口，这是我在 Item 18 中安排的主题。

Things to Remember

- 为了防止资源泄漏，使用 RAII 对象，在 RAII 对象的构造函数中获得资源并在析构函数中释放它们。

- 两个通用的 RAII 是 tr1::shared_ptr 和 auto_ptr。tr1::shared_ptr 通常是更好的选择，因为它的拷贝时的行为是符合直觉的。拷贝一个 auto_ptr 是将它置为空。

Item 14: 谨慎考虑资源管理类的拷贝行为

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 13 介绍了作为资源管理类支柱的 Resource Acquisition Is Initialization (RAII) 原则，并描述了 auto_ptr 和 tr1::shared_ptr 在基于堆的资源上运用这一原则的表现。然而，并非所有的资源都是基于堆的，对于这样的资源，像 auto_ptr 和 tr1::shared_ptr 这样的智能指针通常就不像 resource handlers（资源管理者）那样合适。在这种情况下，有时，你可能要根据你自己的需要去创建你自己的资源管理类。

例如，假设你使用 C API 提供的 lock 和 unlock 函数去操纵 Mutex 类型的互斥体对象：

```
void lock(Mutex *pm); // lock mutex pointed to by pm

void unlock(Mutex *pm); // unlock the mutex
```

为了确保你从不会忘记解锁一个被你加了锁的 Mutex，你希望创建一个类来管理锁。RAII 原则规定了这样一个类的基本结构，通过构造函数获取资源并通过析构函数释放它：

```
class Lock {
public:
  explicit Lock(Mutex *pm)
  : mutexPtr(pm)
  { lock(mutexPtr); }                         // acquire resource

  ~Lock() { unlock(mutexPtr); }               // release resource

private:
  Mutex *mutexPtr;
};
```

客户按照 RAII 风格的惯例来使用 Lock：

```
Mutex m;                    // define the mutex you need to use
...
{                           // create block to define critical section
 Lock ml(&m);               // lock the mutex
...                         // perform critical section operations

}                           // automatically unlock mutex at end
                            // of block
```

这没什么问题，但是如果一个 Lock 对象被拷贝应该发生什么？

```
Lock ml1(&m);                       // lock m

Lock ml2(ml1);                      // copy ml1 to ml2—what should
                                    // happen here?
```

这是一个更一般问题的特定实例，每一个 RAII 类的作者都要面临这样的问题：当一个 RAII 对象被拷贝的时候应该发生什么？大多数情况下，你可以从下面各种可能性中挑选一个：

- 禁止拷贝。在很多情况下，允许 RAII 被拷贝是没有意义的。这对于像 Lock 这样类很可能是正确的，因为同步的基本要素的“副本”很少有什么意义。当拷贝对一个 RAII 类没有什么意义的时候，你应该禁止它。Item 6 解释了如何做到这一点。声明拷贝操作为私有。对于 Lock，看起来也许像这样：

  ```
  class Lock: private Uncopyable {            // prohibit copying — see
  public:                                     // Item 6
  ...                                        // as before
  };
  ```

- 对底层的资源引用计数。有时人们需要的是保持一个资源直到最后一个使用它的对象被销毁。在这种情况下，拷贝一个 RAII 对象应该增加引用这一资源的对象的数目。这也就是使用 tr1::shared_ptr 时“拷贝”的含意。

  通常，RAII 类只需要包含一个 tr1::shared_ptr 数据成员就能够实现引用计数的拷贝行为。例如，如果 Lock 要使用引用计数，他可能要将 mutexPtr 的类型从 Mutex* 改变为 tr1::shared_ptr<Mutex>。不幸的是，tr1::shared_ptr 的缺省行为是当它所指向的东西的引用计数变为 0 的时候将它删除，但这不是我们要的。当我们使用 Mutex 完毕后，我们想要将它解锁，而不是将它删除。

  幸运的是，tr1::shared_ptr 允许一个 "deleter" 规范——当引用计数变为 0 时调用的一个函数或者函数对象。（这一功能是 auto_ptr 所没有的，auto_ptr 总是删除它的指针。）deleter 是 tr1::shared_ptr 的构造函数的可选的第二个参数，所以，代码看起来就像这样：

  ```
  class Lock {
  public:
   explicit Lock(Mutex *pm)       // init shared_ptr with the Mutex
   : mutexPtr(pm, unlock)         // to point to and the unlock func
   {                              // as the deleter

     lock(mutexPtr.get());        // see Item 15 for info on "get"
   }
  private:
   std::tr1::shared_ptr<Mutex> mutexPtr;    // use shared_ptr
  };
  ```

  在这个例子中，注意 Lock 类是如何不再声明一个析构函数的。那是因为它不再需要。Item 5 解释了一个类的析构函数（无论它是编译器生成还是用户定义）会自动调用这个类的非静态（non-static）数据成员的析构函数。在本例中，就是 mutexPtr。但是，当互

斥体的引用计数变为 0 时，mutexPtr 的析构函数会自动调用的是 tr1::shared_ptr 的 deleter ——在此就是 unlock。（看过这个类的源代码的人多半意识到需要增加一条注释表明你并非忘记了析构，而只是依赖编译器生成的缺省行为。）

- 拷贝底层的资源。有时就像你所希望的你可以拥有一个资源的多个副本，唯一的前提是你需要一个资源管理类确保当你使用完它之后，每一副本都会被释放。在这种情况下，拷贝一个资源管理对象也要同时拷贝被它隐藏的资源。也就是说，拷贝一个资源管理类需要完成一次“深层拷贝”。

  某些标准 string 类型的实现是由堆内存的指针组成，堆内存中存储着组成那个 string 的字符。这样的字符串对象包含指向堆内存的指针。当一个 string 对象被拷贝，这个副本应该由那个指针和它所指向的内存组成。这样的 strings 表现为深层拷贝。

- 传递底层资源的所有权。在某些特殊场合，你可能希望确保只有一个 RAII 对象引用一个裸资源（raw resource），而当这个 RAII 对象被拷贝的时候，资源的所有权从被拷贝的对象传递到拷贝对象。就像 Item 13 所说明的，这就是使用 auto_ptr 时“拷贝”的含意。

  拷贝函数（copying functions）（拷贝构造函数和拷贝赋值运算符）可能是由编译器生成的，所以除非编译器生成的版本所做的事正是你所要的（Item 5 说明了这些缺省行为），你应该自己编写它们。在某些情况下，你也要支持这些函数的泛型化版本。这样的版本在 Item 45 描述。

Things to Remember

拷贝一个 RAII 对象必须拷贝它所管理的资源，所以资源的拷贝行为决定了 RAII 对象的拷贝行为。

普通的 RAII 类的拷贝行为不接受拷贝和进行引用计数，但是其它行为是有可能的。

Item 15: 在资源管理类中准备访问裸资源（raw resources）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

资源管理类真是太棒了。他们是你防御资源泄漏的防波堤，没有这样的泄漏是设计良好的系统的基本特征。在一个完美的世界中，你可以在所有与资源的交互中依赖这样的类，从来不需要因为直接访问裸资源（raw resources）而玷污你的手。但是这个世界并不完美，很多 API 直接涉及资源，所以除非你计划坚决放弃使用这样的 API（这种事很少会成为实际），否则，你就要经常绕过资源管理类而直接处理裸资源（raw resources）。

例如，Item 13 介绍的使用类似 auto_ptr 或 tr1::shared_ptr 这样的智能指针来持有调用类似 createInvestment 这样的 factory 函数的结果：

```
std::tr1::shared_ptr&lt;Investment&gt; pInv(createInvestment()); // from Item 13
```

假设你打算使用的一个与 Investment 对象一起工作的函数是这样的：

```
int daysHeld(const Investment *pi);        // return number of days
                                           // investment has been held
```

你打算像这样调用它，

```
int days = daysHeld(pInv); // error!
```

但是这代码不能编译：daysHeld 要求一个裸的 Investment* 指针，但是你传给它一个类型为 tr1::shared_ptr<Investment> 的对象。

你需要一个将 RAII 类（当前情况下是 tr1::shared_ptr）的对象转化为它所包含的裸资源（例如，底层的 Investment*）的方法。有两个常规方法来做这件事。显式转换和隐式转换。

tr1::shared_ptr 和 auto_ptr 都提供一个 get 成员函数进行显示转换，也就是说，返回一个智能指针对象内部的裸指针（raw pointer）（或它的一个副本）：

```
int days = daysHeld(pInv.get());           // fine, passes the raw pointer
                                           // in pInv to daysHeld
```

就像实际上的所有智能指针类一样，tr1::shared_ptr 和 auto_ptr 也都重载了指针解引用操作符（pointer dereferencing operators）（operator-> 和 operator*），而这样就允许隐式转换到底层的裸指针（raw pointers）：

```
class Investment {                                // root class for a hierarchy
public:                                           // of investment types
  bool isTaxFree() const;
  ...
};

Investment* createInvestment();                   // factory function

std::tr1::shared_ptr<Investment>                  // have tr1::shared_ptr
  pi1(createInvestment());                        // manage a resource

bool taxable1 = !(pi1->isTaxFree());              // access resource
                                                  // via operator->
...
std::auto_ptr<Investment> pi2(createInvestment()); // have auto_ptr
                                                  // manage a
                                                  // resource

bool taxable2 = !((*pi2).isTaxFree());            // access resource
                                                  // via operator*

...
```

因为有些时候有必要取得 RAII 对象内部的裸资源，所以一些 RAII 类的设计者就通过提供一个隐式转换函数来给刹车抹油。例如，考虑以下这个 RAII 类，它要为 C API 提供原始状态的字体资源：

```
FontHandle getFont();                     // from C API—params omitted
                                          // for simplicity

void releaseFont(FontHandle fh);          // from the same C API
class Font {                              // RAII class
public:
  explicit Font(FontHandle fh)            // acquire resource;
  : f(fh)                                 // use pass-by-value, because the
  {}                                      // C API does

  ~Font() { releaseFont(f); }             // release resource

private:
  FontHandle f;                           // the raw font resource
};
```

假设有一个巨大的与字体有关的 C API 只能与 FontHandle 打交道，这就需要频繁地将 Font 对象转换为 FontHandle。Font 类可以提供一个显式的转换函数，比如 get：

```
class Font {
public:
  ...
  FontHandle get() const { return f; }     // explicit conversion function
  ...
};
```

不幸的是，这就要求客户每次与 API 通信时都要调用 get：

```
void changeFontSize(FontHandle f, int newSize);     // from the C API

Font f(getFont());
int newFontSize;
...

changeFontSize(f.get(), newFontSize);               // explicitly convert
                                                    // Font to FontHandle
```

一些程序员可能发现对显式请求这个转换的需求足以令人郁闷而避免使用这个类。反过来，设计 Font 类又是为了预防泄漏字体资源的机会的增长。

可选择的办法是为 Font 提供一个隐式转换到它的 FontHandle 的转换函数：

```
class Font {
public:
  ...
  operator FontHandle() const { return f; }        // implicit conversion function
  ...
};
```

这样就可以使对 C API 的调用简单而自然：

```
Font f(getFont());
int newFontSize;
...

changeFontSize(f, newFontSize);             // implicitly convert Font
                                            // to FontHandle
```

不利的方面是隐式转换增加了错误的机会。例如，一个客户可能会在有意使用 Font 的地方意外地产生一个 FontHandle：

```
Font f1(getFont());

...

FontHandle f2 = f1;                         // oops! meant to copy a Font
                                            // object, but instead implicitly
                                            // converted f1 into its underlying
                                            // FontHandle, then copied that
```

现在，程序有了一个被 Font 对象 f1 管理的 FontHandle，但是这个 FontHandle 也能通过直接使用 f2 来加以利用。这几乎绝对不会成为什么好事。例如，当 f1 被销毁，字体将被释放，f2 则被悬挂（dangle）。

关于是否提供从一个 RAII 类到它的底层资源的显式转换（例如，通过一个 get 成员函数）或者允许隐式转换的决定，要依靠 RAII 类被设计履行的具体任务和它被计划使用的细节而做出。最好的设计很可能就是坚持 Item 18 的建议（使接口易于正确使用，而难以错误使用）的那一个。通常，类似 get 的一个显式转换函数是更可取的方式，因为它将意外的类型转换的机会减到最少。偶尔的，通过隐式类型转换提高使用的自然性将使天平向那个方向倾斜。

你可能已经意识到，函数返回一个 RAII 类内部的裸资源破坏了封装。这是正确的，但这并非像它开始看上去那样是个设计的祸患。RAII 类的存在并非为了封装什么东西；它的存在是为了确保一个特殊的动作——资源释放——的发生。如果你希望，封装资源的地位也可以提高到这个主要功能之上，但这并非必需。此外，一些 RAII 类将实现的真正封装和底层资源的非常宽松的封装结合在一起。例如，tr1::shared_ptr 封装了它的引用计数的全部机制，但它依然提供对它所包含的资源的简单访问。就像大多数设计良好的类，它隐藏了客户不需要看到的，但它也让客户的确需要访问的那些东西可以利用。

Things to Remember

- API 经常需要访问裸资源，所以每一个 RAII 类都应该提供取得它所管理的资源的方法。
- 访问可以通过显式转换或者隐式转换进行。通常，显式转换更安全，而隐式转换对客户来说更方便。

Item 16: 使用相同形式的 new 和 delete

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

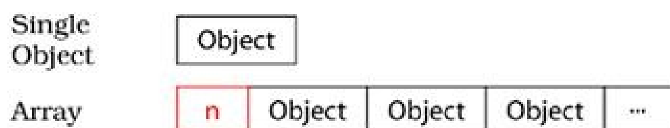
下面这段代码有什么问题？

```
std::string *stringArray = new std::string[100];
...
delete stringArray;
```

每件事看起来都很正常。也为 new 搭配了一个 delete。但是，仍然有某件事情彻底错了。程序的行为是未定义的。直到最后，stringArray 指向的 100 个 string 对象中的 99 个不太可能被完全销毁，因为它们的析构函数或许根本没有被调用。

当你使用了一个 new 表达式（也就是说，通过使用 new 动态创建一个对象），有两件事情会发生。首先，分配内存（通过一个被称为 operator new 的函数——参见 Item 49 和 51）。第二，一个或多个构造函数在这些内存上被调用。当你使用一个 delete 表达式（也就是说，使用 delete），有另外的两件事情会发生：一个或多个析构函数在这些内存上被调用，然后内存被回收（通过一个被称为 operator delete 的函数——参见 Item 51）。对于 delete 来说有一个大问题：在要被删除的内存中到底驻留有多少个对象？这个问题的答案将决定有多少个析构函数必须被调用。

事实上，问题很简单：将要被删除的指针是指向一个单一的对象还是一个对象的数组？这是一个关键的问题，因为单一对象的内存布局通常不同于数组的内存布局。详细地说，一个数组的内存布局通常包含数组的大小，这样可以使得 delete 更容易知道有多少个析构函数需要被调用。而一个单一对象的内存中缺乏这个信息。你可以认为不同的内存布局看起来如下图，那个 n 就是数组的大小：



这当然只是一个例子。编译器并不是必须这样实现，虽然很多是这样的。

当你对一个指针使用 delete，delete 知道是否有数组大小信息的唯一方法就是由你来告诉它。如果你在你使用的 delete 中加入了方括号，delete 就假设那个指针指向的是一个数组。否则，就假设指向一个单一的对象。

```
std::string *stringPtr1 = new std::string;

std::string *stringPtr2 = new std::string[100];

...

delete stringPtr1;                     // delete an object

delete [] stringPtr2;                  // delete an array of objects
```

如果你对 stringPtr1 使用了 [] 形式会发生什么呢？结果是未定义的，但不太可能是什么好事。假设如上图的布局，delete 将读入某些内存的内容并将其看作一个数组的大小，然后开始调用那么多析构函数，不仅全然不顾它在其上工作的内存不是数组，而且还可能忘掉了它正忙着析构的对象的类型。

如果你对 stringPtr2 没有使用 [] 形式会发生什么呢？也是未定义的，只不过你不会看到它会引起过多的析构函数被调用。此外，对于类似 int 的内建类型其结果也是未定义的（而且有时是有害的），即使这样的类型没有析构函数。

规则很简单。如果你在 new 表达式中使用了 []，你也必须在相应的 delete 表达式中使用 []。如果你在 new 表达式中没有使用 []，在匹配的 delete 表达式中也不要使用 []。

当你写的一个类中包含一个指向动态分配的内存的指针，而且提供了多个构造函数的时候，这条规则尤其重要，应镌刻脑海，因为那时你必须小心地在所有的构造函数中使用相同形式的 new 初始化那个指针成员。如果你不这样做，你怎么知道在你的析构函数中应该使用哪种形式的 delete 呢？

这个规则对于有 typedef 倾向的人也很值得注目，因为这意味着一个 typedef 的作者必须在文档中记录：当用 new 生成一个 typedef 类型的对象时，应该使用哪种形式的 delete。例如，考虑这个 typedef：

```
typedef std::string AddressLines[4];   // a person's address has 4 lines,
                                       // each of which is a string
```

因为 AddressLines 是一个数组，这里使用 new，

```
std::string *pal = new AddressLines;   // note that "new AddressLines"
                                       // returns a string*, just like
                                       // "new string[4]" would
```

必须用 delete 的数组形式进行匹配：

```
delete pal; // undefined!

delete [] pal; // fine
```

为了避免这种混淆，要克制对数组类型使用 typedef。那很简单，因为标准 C++ 库（参见 Item 54）包含 string 和 vector，而且那些模板将对动态分配数组的需要减少到几乎为零。例如，这里，AddressLines 可以被定义为一个 string 的 vector，也就是说，类型为 vector<string>。

Things to Remember

- 如果你在 new 表达式中使用了 []，你必须在对应的 delete 表达式中使用 []。如果你在 new 表达式中没有使用 []，你也不必在对应的 delete 表达式中不使用 []。

Item 17: 在一个独立的语句中将 new 出来的对象存入智能指针

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们有一个函数取得我们的处理优先级，而第二个函数根据优先级针对动态分配的 Widget 做一些处理：

```
int priority();

void processWidget(std::tr1::shared_ptr&lt;Widget&gt; pw, int priority);
```

不要忘记使用对象管理资源的至理名言（参见 Item 13），processWidget 为处理动态分配的 Widget 使用了一个智能指针（在此，是一个 tr1::shared_ptr）。

现在考虑一个对 processWidget 的调用：

```
processWidget(new Widget, priority());
```

且慢，别想这样调用。它不能编译。tr1::shared_ptr 的构造函数取得一个裸指针（raw pointer）应该是显式的，所以不能从一个由 "new Widget" 返回的裸指针隐式转型到 processWidget 所需要的 tr1::shared_ptr。下面的代码，无论如何，是可以编译的：

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

令人惊讶的是，尽管我们在这里各处都使用了对象管理资源，这个调用还是可能泄漏资源。下面就来说明这是如何发生的。

在编译器能生成一个对 processWidget 的调用之前，它们必须传递实际参数来计算形式参数的值。第二个实际参数不过是对函数 priority 的调用，但是第一个实际参数（"std::tr1::shared_ptr<Widget>(new Widget)"），由两部分组成

- 表达式 "new Widget" 的执行。
- 一个对 tr1::shared_ptr 的构造函数的调用。

在 processWidget 能被调用之前，编译器必须为这三件事情生成代码：

- 调用 priority。

- 执行 "new Widget"。
- 调用 tr1::shared_ptr 的构造函数。

C++ 编译器允许在一个相当大的范围内决定这三件事被完成的顺序。（这里与 Java 和 C# 等语言的处理方式不同，那些语言里函数参数总是按照一个精确的顺序被计算。）"new Widget" 表达式一定在 tr1::shared_ptr 的构造函数能被调用之前执行，因为这个表达式的结果要作为一个参数传递给 tr1::shared_ptr 的构造函数，但是 priority 的调用可以被第一个，第二个或第三个执行。如果编译器选择第二个执行它（大概这样能使它们生成更有效率的代码），我们最终得到这样一个操作顺序：

1. 执行 "new Widget"。

2. 调用 priority。

3. 调用 tr1::shared_ptr 的构造函数。

但是请考虑，如果对 priority 的调用引发一个异常将发生什么。在这种情况下，从 "new Widget" 返回的指针被丢失，因为它没有被存入我们期望能阻止资源泄漏的 tr1::shared_ptr。由于一个异常可能插入资源创建的时间和将资源交给一个资源管理对象的时间之间，所以调用 processWidget 可能会发生一次泄漏。

避免类似问题的方法很简单：用一个单独的语句创建 Widget 并将它存入一个智能指针，然后将这个智能指针传递给 processWidget：

```
std::tr1::shared_ptr<Widget> pw(new Widget);  // store newed object
                                              // in a smart pointer in a
                                              // standalone statement

processWidget(pw, priority());                // this call won't leak
```

这样做是因为编译器在不同的语句之间重新安排操作顺序的活动余地比在一个语句之内要小得多。"new Widget" 表达式和 tr1::shared_ptr 的构造函数的调用与 priority 的调用在不同的语句中，所以编译器不会允许 priority 的调用插入它们中间。

Things to Remember

- 在一个独立的语句中将 new 出来的对象存入智能指针。如果疏忽了这一点，当异常发生时，可能引起微妙的资源泄漏。

Item 18: 使接口易于正确使用，而难以错误使用

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 被淹没于接口中。函数接口、类接口、模板接口。每一个接口都意味着客户的代码和你的代码互相影响。假设你在和通情达理的人打交道，那些客户也想做好工作。他们想要正确使用你的接口。在这种情况下，如果他们犯了一个错误，就说明你的接口至少有部分是不完善的。在理想情况下，如果一个接口的一种尝试的用法不符合客户的预期，代码将无法编译，反过来，如果代码可以编译，那么它做的就是客户想要的。

开发易于正确使用，而难以错误使用的接口需要你考虑客户可能造成的各种错误。例如，假设你正在设计一个代表时间的类的构造函数：

```
class Date {
public:
  Date(int month, int day, int year);
  ...
};
```

匆匆一看，这个接口似乎是合乎情理的（至少在美国），但是客户可能很容易地造成两种错误。首先，他们可能会以错误的顺序传递参数：

```
Date d(30, 3, 1995); // Oops! Should be "3, 30" , not "30, 3"
```

第二，他们可能传递一个非法的代表月或日的数字：

```
Date d(2, 20, 1995); // Oops! Should be "3, 30" , not "2, 20"
```

（后面这个例子看上去好像没什么，但是想想键盘上，2 就在 3 的旁边，这种 "off by one" 类型的错误并不罕见。）

很多客户错误都可以通过引入新的类型来预防。确实，类型系统是你阻止那些不合适的代码通过编译的主要支持者。在当前情况下，我们可以引入简单的包装类型来区别日，月和年，并将这些类型用于 Data 的构造函数。

```
struct Day {            struct Month {              struct Year {
  explicit Day(int d)     explicit Month(int m)       explicit Year(int y)
  :val(d) {}              :val(m) {}                  :val(y){}

  int val;                int val;                    int val;
};                      };                          };

class Date {
public:
 Date(const Month& m, const Day& d, const Year& y);
 ...
};
Date d(30, 3, 1995);                      // error! wrong types

Date d(Day(30), Month(3), Year(1995));    // error! wrong types

Date d(Month(3), Day(30), Year(1995));    // okay, types are correct
```

将日，月和年做成封装数据的羽翼丰满的类比上面的简单地使用 struct 更好（参见 Item 22），但是即使是 struct 也足够证明明智地引入新类型在阻止接口的错误使用方面能工作得非常出色。

只要放置了正确的类型，它往往能合理地限制那些类型的值。例如，月仅有 12 个合法值，所以 Month 类型应该反映这一点。做到这一点的一种方法是用一个枚举来表现月，但是枚举不像我们希望的那样是类型安全（type-safe）的。例如，枚举能被作为整数使用（参见 Item 2）。一个安全的解决方案是预先确定合法的 Month 的集合：

```
class Month {
public:
  static Month Jan() { return Month(1); }   // functions returning all valid
  static Month Feb() { return Month(2); }   // Month values; see below for
  ...                                       // why these are functions, not
  static Month Dec() { return Month(12); }  // objects

  ...                                       // other member functions

private:
  explicit Month(int m);                    // prevent creation of new
                                            // Month values

  ...                                       // month-specific data
};
Date d(Month::Mar(), Day(30), Year(1995));
```

如果用函数代替对象来表现月的主意让你感到惊奇，那可能是因为你忘了非局部静态对象（non-local static objects）的初始化的可靠性是值得怀疑的。Item 4 能唤起你的记忆。

防止可能的客户错误的另一个方法是限制对一个类型能够做的事情。施加限制的一个普通方法就是加上 const。例如，Item 3 解释了使 operator* 的返回类型具有 const 资格是如何能够防止客户对用户自定义类型犯下这样的错误：

```
if (a * b = c) ... // oops, meant to do a comparison!
```

实际上，这仅仅是另一条使类型易于正确使用而难以错误使用的普遍方针的一种表现：除非你有很棒的理由，否则就让你的类型的行为与内建类型保持一致。客户已经知道像 int 这样的类型如何表现，所以你应该努力使你的类型的表现无论何时都同样合理。例如，如果 a 和 b 是 int，给 a*b 赋值是非法的。所以除非有一个非常棒理由脱离这种表现，否则，对你的类型来说这样做也应该是非法的。

避免和内建类型毫无理由的不相容的真正原因是为了提供行为一致的接口。很少有特性比一致性更易于引出易于使用的接口，也很少有特性比不一致性更易于引出令人郁闷的接口。STL 容器的接口在很大程度上（虽然并不完美）是一致的，而且这使得它们相当易于使用。例如，每一种 STL 容器都有一个名为 size 的成员函数可以知道容器中有多少对象。与此对比的是 Java，在那里你对数组使用 length 属性，对 String 使用 length 方法，而对 List 却要使用 size 方法，在 .NET 中，Array 有一个名为 Length 的属性，而 ArrayList 却有一个名为 Count 的属性。一些开发人员认为集成开发环境（IDEs）能补偿这些琐细的矛盾，但他们错了。矛盾在开发者工作中强加的精神折磨是任何 IDE 都无法完全消除的。

任何一个要求客户记住某些事情的接口都是有错误使用倾向的，因为客户可能忘记做那些事情。例如，Item 13 介绍了一个 factory 函数，它返回一个指向动态分配的 Investment 继承体系中的对象的指针。

```
Investment* createInvestment();       // from Item 13; parameters omitted
                                      // for simplicity
```

为了避免资源泄漏，createInvestment 返回的指针最后必须被删除，但这就为至少两种类型的客户错误创造了机会：删除指针失败，或删除同一个指针一次以上。

Item 13 展示了客户可以怎样将 createInvestment 的返回值存入一个类似 auto_ptr 或 tr1::shared_ptr 智能指针，从而将使用 delete 的职责交给智能指针。但是如果客户忘记使用智能指针呢？在很多情况下，一个更好的接口会预先判定将要出现的问题，从而让 factory 函数在第一现场即返回一个智能指针：

```
std::tr1::shared_ptr<Investment> createInvestment();
```

这就从根本上强制客户将返回值存入一个 tr1::shared_ptr，几乎完全消除了当底层的 Investment 对象不再使用的时候忘记删除的可能性。

实际上，返回一个 tr1::shared_ptr 使得接口的设计者预防许多其它客户的与资源泄漏相关的错误成为可能，因为，就像 Item 14 解释的：当一个智能指针被创建的时候，tr1::shared_ptr 允许将一个资源释放（resource-release）函数——一个 "deleter" ——绑定到智能指针上。（auto_ptr 则没有这个能力。）

假设从 createInvestment 得到一个 Investment* 指针的客户期望将这个指针传给一个名为 getRidOfInvestment 的函数，而不是对它使用 delete。这样一个接口又为一种新的客户错误打开了门，这就是客户可能使用了错误的资源析构机制（也就是说，用了 delete 而不是

getRidOfInvestment）。createInvestment 的实现可以通过返回一个在它的 deleter 上绑定了 getRidOfInvestment 的 tr1::shared_ptr 来预防这个问题。

tr1::shared_ptr 提供了一个需要两个参数（要被管理的指针和当引用计数变为零时要调用的 deleter）的构造函数。这里展示了创建一个以 getRidOfInvestment 为 deleter 的 null tr1::shared_ptr 的方法：

```
std::tr1::shared_ptr<Investment>      // attempt to create a null
  pInv(0, getRidOfInvestment);        // shared_ptr with a custom deleter;
                                      // this won't compile
```

唉，这不是合法的 C++。tr1::shared_ptr 的构造函数坚决要求它的第一个参数应该是一个指针，而 0 不是一个指针，它是一个 int。当然，它能转型为一个指针，但那在当前情况下并不够好用，tr1::shared_ptr 坚决要求一个真正的指针。用强制转型解决这个问题：

```
std::tr1::shared_ptr<Investment>      // create a null shared_ptr with
  pInv(static_cast<Investment*>(0),   // getRidOfInvestment as its
         getRidOfInvestment);         // deleter; see Item 27 for info on
                                      // static_cast
```

据此，实现返回一个以 getRidOfInvestment 作为 deleter 的 tr1::shared_ptr 的 createInvestment 的代码看起来就像这个样子：

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  std::tr1::shared_ptr<Investment> retVal(static_cast<Investment*>(0),
                                          getRidOfInvestment);

  retVal = ... ;                         // make retVal point to the
                                         // correct object

  return retVal;
}
```

当然，如果将被 pInv 管理的裸指针可以在创建 pInv 时被确定，最好是将这个裸指针传给 pInv 的构造函数，而不是将 pInv 初始化为 null 然后再赋值给它。至于方法上的细节，参考 Item 26。

tr1::shared_ptr 的一个特别好的特性是它自动逐指针地使用 deleter 以消除另一种潜在的客户错误——“cross-DLL 问题。”这个问题发生在这种情况下：一个对象在一个动态链接库（dynamically linked library (DLL)）中通过 new 被创建，在另一个不同的 DLL 中被删除。在许多平台上，这样的 cross-DLL new/delete 对会引起运行时错误。tr1::shared_ptr 可以避免这个问题，因为它的缺省的 deleter 只将 delete 用于这个 tr1::shared_ptr 被创建的 DLL 中。这就意味着，例如，如果 Stock 是一个继承自 Investment 的类，而且 createInvestment 被实现如下，

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  return std::tr1::shared_ptr<Investment>(new Stock);
}
```

返回的 tr1::shared_ptr 能在 DLL 之间进行传递，而不必关心 cross-DLL 问题。指向这个 Stock 的 tr1::shared_ptr 将保持对“当这个 Stock 的引用计数变为零的时候，哪一个 DLL 的 delete 应该被使用”的跟踪。

这个 Item 不是关于 tr1::shared_ptr 的——而是关于使接口易于正确使用，而难以错误使用的——但 tr1::shared_ptr 正是这样一个消除某些客户错误的简单方法，值得用一个概述来看看使用它的代价。最通用的 tr1::shared_ptr 实现来自于 Boost（参见 Item 55）。Boost 的 shared_ptr 的大小是裸指针的两倍，将动态分配内存用于簿记和 deleter 专用（deleter-specific）数据，当调用它的 deleter 时使用一个虚函数来调用，在一个它认为是多线程的应用程序中，当引用计数被改变，会导致线程同步开销。（你可以通过定义一个预处理符号来使多线程支持失效。）在缺点方面，它比一个裸指针大，比一个裸指针慢，而且要使用辅助的动态内存。在许多应用程序中，这些附加的运行时开销并不显著，而对客户错误的减少却是每一个人都看得见的。

Things to Remember

- 好的接口易于正确使用，而难以错误使用。你应该在你的所有接口中为这个特性努力。
- 使易于正确使用的方法包括在接口和行为兼容性上与内建类型保持一致。
- 预防错误的方法包括创建新的类型，限定类型的操作，约束对象的值，以及消除客户的资源管理职责。
- tr1::shared_ptr 支持自定义 deleter。这可以防止 cross-DLL 问题，能用于自动解锁互斥体（参见 Item 14）等。

Item 19: 视类设计为类型设计

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在 C++ 中，就像其它面向对象编程语言，可以通过定义一个新的类来定义一个新的类型。作为一个 C++ 开发者，你的大量时间就这样花费在增大你的类型系统。这意味着你不仅仅是一个类的设计者，而且是一个类型的设计者。重载函数和运算符，控制内存分配和回收，定义对象的初始化和终结过程——这些全在你的掌控之中。因此你应该在类设计中倾注大量心血，接近语言设计者在语言内建类型的设计中所倾注的大量心血。

设计良好的类是有挑战性的，因为设计良好的类型是有挑战性的。良好的类型拥有简单自然的语法，符合直觉的语义，以及一个或更多高效的实现。在 C++ 中，一个缺乏计划的类设计，使其不可能达到上述任何一个目标。甚至一个类的成员函数的执行特性可能受到它们是被如何声明的影响。

那么，如何才能设计高效的类呢？首先，你必须理解你所面对的问题。实际上每一个类都需要你面对下面这些问题，其答案通常就导向你的设计的限制因素：

- 你的新类型的对象应该如何创建和销毁？如何做这些将影响到你的类的构造函数和析构函数，以及内存分配和回收的函数（operator new，operator new[]，operator delete，和 operator delete[] ——参见 Chapter 8）的设计，除非你不写它们。

- 对象的初始化和对象的赋值应该有什么不同？这个问题的答案决定了你的构造函数和你的赋值运算符的行为和它们之间的不同。这对于不混淆初始化和赋值是很重要的，因为它们相当于不同的函数调用（参见 Item 4）。

- 以值传递（passed by value）对于你的新类型的对象意味着什么？记住，拷贝构造函数定义了一个新类型的传值（pass-by-value）如何实现。

- 你的新类型的合法值的限定条件是什么？通常，对于一个类的数据成员来说，仅有某些值的组合是合法的。那些组合决定了你的类必须维持的不变量。这些不变量决定了你必须在成员函数内部进行错误检查，特别是你的构造函数，赋值运算符，以及 "setter" 函数。它可能也会影响你的函数抛出的异常，以及你的函数的异常规范（exception specification）（你用到它的可能性很小）。

- 你的新类型是否适合放进一个继承图表中？如果你从已经存在的类继承，你将被那些类的设计所约束，特别是它们的函数是 virtual 还是 non-virtual（参见 Item 34 和 36）。如果你希望允许其他类继承你的类，将影响到你是否将函数声明为 virtual，特别是你的析构函数（参见 Item 7）。

- 你的新类型允许哪种类型转换？你的类型身处其它类型的海洋中，所以是否要在你的类型和其它类型之间有一些转换？如果你希望允许 T1 类型的对象隐式转型为 T2 类型的对象，你就要么在 T1 类中写一个类型转换函数（例如，operator T2），要么在 T2 类中写一个非显式的构造函数，而且它们都要能够以单一参数调用。如果你希望仅仅允许显示转换，你就要写执行这个转换的函数，而且你还需要避免使它们的类型转换运算符或非显式构造函数能够以一个参数调用。（作为一个既允许隐式转换又允许显式转换的例子，参见 Item 15。）

- 对于新类型哪些运算符和函数有意义？这个问题的答案决定你应该为你的类声明哪些函数。其中一些是成员函数，另一些不是（参见 Item 23、24 和 46）。

- 哪些标准函数不应该被接受？你需要将那些都声明为 private（参见 Item 6）。

- 你的新类型中哪些成员可以被访问？这个问题的可以帮助你决定哪些成员是 public，哪些是 protected，以及哪些是 private。它也可以帮助你决定哪些类和／或函数应该是友元，以及一个类嵌套在另一个类内部是否有意义。

- 什么是你的新类型的 "undeclared interface"？它对于性能考虑，异常安全（exception safety）（参见 Item 29），以及资源使用（例如，锁和动态内存）提供哪种保证？你在这些领域提供的保证将强制影响你的类的实现。

- 你的新类型有多大程度的通用性？也许你并非真的要定义一个新的类型。也许你要定义一个整个的类型家族。如果是这样，你不需要定义一个新的类，而是需要定义一个新的类模板。

- 一个新的类型真的是你所需要的吗？是否你可以仅仅定义一个新的继承类，以便让你可以为一个已存在的类增加一些功能，也许通过简单地定义一个或更多非成员函数或模板能更好地达成你的目标。

回答这些问题是困难的，所以定义高效的类是有挑战性的。既然，在 C++ 中用户自定义类生成的类型至少可以和内建类型一样好，那就做好它，它会使一切努力都变的有价值。

Things to Remember

- 类设计就是类型设计。定义一个新类型之前，确保考虑了本 Item 讨论的所有问题。

Item 20: 用 pass-by-reference-to-const（传引用给 const）取代 pass-by-value（传值）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

缺省情况下，C++ 以传值方式将对象传入或传出函数（这是一个从 C 继承来的特性）。除非你特别指定其它方式，否则函数的参数就会以实际参数（actual argument）的拷贝进行初始化，而函数的调用者会收到函数返回值的一个拷贝。这个拷贝由对象的拷贝构造函数生成。这就使得传值（pass-by-value）成为一个代价不菲的操作。例如，考虑下面这个类层级结构：

```
class Person {
public:
  Person();                          // parameters omitted for simplicity
  virtual ~Person();                 // see Item 7 for why this is virtual
  ...

private:
  std::string name;
  std::string address;
};

class Student: public Person {
public:
  Student();                         // parameters again omitted
  virtual ~Student();
  ...

private:
  std::string schoolName;
  std::string schoolAddress;
};
```

现在，考虑以下代码，在此我们调用一个函数—— validateStudent，它得到一个 Student 参数（以传值的方式），并返回它是否验证有效的结果：

```
bool validateStudent(Student s);           // function taking a Student
                                           // by value

Student plato;                             // Plato studied under Socrates

bool platoIsOK = validateStudent(plato);   // call the function
```

当这个函数被调用时会发生什么呢？

很明显，Student 的拷贝构造函数被调用，用 plato 来初始化参数 s。同样明显的是，当 validateStudent 返回时，s 就会被销毁。所以这个函数的参数传递代价是一次 Student 的拷贝构造函数的调用和一次 Student 的析构函数的调用。

但这还不是全部。一个 Student 对象内部包含两个 string 对象，所以每次你构造一个 Student 对象的时候，你也必须构造两个 string 对象。一个 Student 对象还要从一个 Person 对象继承，所以每次你构造一个 Student 对象的时候，你也必须构造一个 Person 对象。一个 Person 对象内部又包含两个额外的 string 对象，所以每个 Person 的构造也承担着另外两个 string 的构造。最终，以传值方式传递一个 Student 对象的后果就是引起一次 Student 的拷贝构造函数的调用，一次 Person 的拷贝构造函数的调用，以及四次 string 的拷贝构造函数调用。当 Student 对象的拷贝被销毁时，每一个构造函数的调用都对应一个析构函数的调用，所以以传值方式传递一个 Student 的全部代价是六个构造函数和六个析构函数！

好了，这是正确的和值得的行为。毕竟，你希望你的全部对象都得到可靠的初始化和销毁。尽管如此，如果有一种办法可以绕过所有这些构造和析构过程，应该变得更好，这就是：传引用给 const（pass by reference-to-const）：

```
bool validateStudent(const Student& s);
```

这样做非常有效：没有任何构造函数和析构函数被调用，因为没有新的对象被构造。被修改的参数声明中的 const 是非常重要的。validateStudent 的最初版本接受一个 Student 值参数，所以调用者知道它们屏蔽了函数对它们传入的 Student 的任何可能的改变；validateStudent 也只能改变它的一个拷贝。现在 Student 以引用方式传递，同时将它声明为 const 是必要的，否则调用者必然担心 validateStudent 改变了它们传入的 Student。

以传引用方式传递参数还可以避免切断问题（slicing problem）。当一个派生类对象作为一个基类对象被传递（传值方式），基类的拷贝构造函数被调用，而那些使得对象的行为像一个派生类对象的特殊特性被“切断”了。你只剩下一个纯粹的基类对象——这没什么可吃惊的，因为是一个基类的构造函数创建了它。这几乎绝不是你希望的。例如，假设你在一组实现一个图形窗口系统的类上工作：

```
class Window {
public:
  ...
  std::string name() const;           // return name of window
  virtual void display() const;       // draw window and contents
};

class WindowWithScrollBars: public Window {
public:
  ...
  virtual void display() const;
};
```

所有 Window 对象都有一个名字，你能通过 name 函数得到它，而且所有的窗口都可以显示，你可一个通过调用 display 函数来做到这一点。display 为 virtual 的事实清楚地告诉你：一个纯粹的基类的 Window 对象的显示方法有可能不同于专门的 WindowWithScrollBars 对象

的显示方法（参见 Item 34 和 36）。

现在，假设你想写一个函数打印出一个窗口的名字，并随后显示这个窗口。以下这个函数的写法是错误的：

```
void printNameAndDisplay(Window w)         // incorrect! parameter
{                                          // may be sliced!
  std::cout << w.name();
  w.display();
}
```

考虑当你用一个 WindowWithScrollBars 对象调用这个函数时会发生什么：

```
WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

参数 w 将被作为一个 Window 对象构造——它是被传值的，记得吗？而且使 wwsb 表现得像一个 WindowWithScrollBars 对象的特殊信息都被切断了。在 printNameAndDisplay 中，全然不顾传递给函数的那个对象的类型，w 将始终表现得像一个 Window 类的对象（因为它就是一个 Window 类的对象）。特别是，在 printNameAndDisplay 中调用 display 将总是调用 Window::display，绝不会是 WindowWithScrollBars::display。

绕过切断问题的方法就是以传引用给 const 的方式传递 w：

```
void printNameAndDisplay(const Window& w)   // fine, parameter won't
{                                           // be sliced
  std::cout << w.name();
  w.display();
}
```

现在 w 将表现得像实际传入的那种窗口。

如果你掀开编译器的盖头偷看一下，你会发现用指针实现引用是非常典型的做法，所以以引用传递某物实际上通常意味着传递一个指针。由此可以得出结论，如果你有一个内建类型的对象（例如，一个 int），以传值方式传递它常常比传引用方式更高效。那么，对于内建类型，当你需要在传值和传引用给 const 之间做一个选择时，没有道理不选择传值。同样的建议也适用于 STL 中的迭代器（iterators）和函数对象（function objects），因为，作为惯例，它们就是为传值设计的。迭代器（iterators）和函数对象（function objects）的实现有责任保证拷贝的高效并且不受切断问题的影响。（这是一个“规则如何变化，依赖于你使用 C++ 的哪一个部分”的实例——参见 Item 1。）

内建类型很小，所以有人就断定所有的小类型都是传值的上等候选者，即使它们是用户定义的。这样的推论是不可靠的。仅仅因为一个对象小，并不意味着调用它的拷贝构造函数就是廉价的。很多对象——大多数 STL 容器也在其中——容纳的和指针一样，但是拷贝这样的对象必须同时拷贝它们指向的每一样东西。那可能是非常昂贵的。

即使当一个小对象有一个廉价的拷贝构造函数，也会存在性能问题。一些编译器对内建类型和用户定义类型并不一视同仁，即使他们有同样的底层表示。例如，一些编译器拒绝将仅由一个 double 组成的对象放入一个寄存器中，即使在常规上它们非常愿意将一个纯粹的 double 放入那里。如果发生了这种事情，你以传引用方式传递这样的对象更好一些，因为编译器理所当然会将一个指针（引用的实现）放入寄存器。

小的用户定义类型不一定是传值的上等候选者的另一个原因是：作为用户定义类型，它的大小常常变化。一个现在较小的类型在将来版本中可能变得更大，因为它的内部实现可能会变化。甚至当你换了一个不同的 C++ 实现时，事情都可能会变化。例如，就在我这样写的时候，一些标准库的 string 类型的实现的大小就是另外一些实现的七倍。

通常情况下，你能合理地假设传值廉价的类型仅有内建类型及 STL 中的迭代器和函数对象类型。对其他任何类型，请遵循本 Item 的建议，并用传引用给 const 取代传值。

Things to Remember

- 用传引用给 const 取代传值。典型情况下它更高效而且可以避免切断问题。
- 这条规则并不适用于内建类型及 STL 中的迭代器和函数对象类型。对于它们，传值通常更合适。

Item 21: 当你必须返回一个对象时不要试图返回一个引用

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

一旦程序员抓住对象传值的效率隐忧（参见 Item 20），很多人就会成为狂热的圣战分子，誓要根除传值的罪恶，无论它隐藏多深。他们不屈不挠地追求传引用的纯度，但他们全都犯了一个致命的错误：他们开始传递并不存在的对象的引用。这可不是什么好事。

考虑一个代表有理数的类，包含一个将两个有理数相乘的函数：

```
class Rational {
public:
  Rational(int numerator = 0,               // see Item 24 for why this
           int denominator = 1);            // ctor isn't declared explicit

  ...

private:
  int n, d;                                 // numerator and denominator

friend
   const Rational                           // see Item 3 for why the
     operator*(const Rational& lhs,         // return type is const
               const Rational& rhs);
};
```

operator* 的这个版本以传值方式返回它的结果，而且如果你没有担心那个对象的构造和析构的代价，你就是在推卸你的专业职责。如果你不是迫不得已，你不应该为这样的一个对象付出成本。所以问题就在这里：你是迫不得已吗？

哦，如果你能用返回一个引用来作为代替，你就不是迫不得已。但是，请记住一个引用仅仅是一个名字，一个实际存在的对象的名字。无论何时只要你看到一个引用的声明，你应该立刻问自己它是什么东西的另一个名字，因为它必定是某物的另一个名字。在这个 operator* 的情况下，如果函数返回一个引用，它必须返回某个已存在的而且其中包含两个对象相乘的产物的 Rational 对象的引用。

当然没有什么理由期望这样一个对象在调用 operator* 之前就存在。也就是说，如果你有

```
Rational a(1, 2);                        // a = 1/2
Rational b(3, 5);                        // b = 3/5

Rational c = a * b;                      // c should be 3/10
```

似乎没有理由期望那里碰巧已经存在一个值为十分之三的有理数。不是这样的，如果 operator* 返回这样一个数的引用，它必须自己创建那个数字对象。

一个函数创建一个新对象仅有两种方法：在栈上或者在堆上。栈上的生成物通过定义一个局部变量而生成。使用这个策略，你可以用这种方法试写 operator*：

```
const Rational& operator*(const Rational& lhs,   // warning! bad code!
                          const Rational& rhs)
{
  Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

你可以立即否决这种方法，因为你的目标是避免调用构造函数，而 result 正像任何其它对象一样必须被构造。一个更严重的问题是这个函数返回一个引向 result 的引用，但是 result 是一个局部对象，而局部对象在函数退出时被销毁。那么，这个 operator* 的版本不会返回引向一个 Rational 的引用——它返回引向一个前 Rational；一个曾经的 Rational；一个空洞的、恶臭的、腐败的，从前是一个 Rational 但永不再是的尸体的引用，因为它已经被销毁了。任何调用者甚至于没有来得及匆匆看一眼这个函数的返回值就立刻进入了未定义行为的领地。这是事实，任何返回一个引向局部变量的引用的函数都是错误的。（对于任何返回一个指向局部变量的指针的函数同样成立。）

那么，让我们考虑一下在堆上构造一个对象并返回引向它的引用的可能性。基于堆的对象通过使用 new 而开始存在，所以你可以像这样写一个基于堆的 operator*：

```
const Rational& operator*(const Rational& lhs,   // warning! more bad
                          const Rational& rhs)   // code!
{
  Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

哦，你还是必须要付出一个构造函数调用的成本，因为通过 new 分配的内存要通过调用一个适当的构造函数进行初始化，但是现在你有另一个问题：谁是删除你用 new 做出来的对象的合适人选？

即使调用者尽职尽责且一心向善，它们也不太可能是用这样的方案来合理地预防泄漏：

```
Rational w, x, y, z;

w = x * y * z;                     // same as operator*(operator*(x, y), z)
```

这里，在同一个语句中有两个 operator* 的调用，因此 new 被使用了两次，这两次都需要使用 delete 来销毁。但是 operator* 的客户没有合理的办法进行那些调用，因为他们没有合理的办法取得隐藏在通过调用 operator* 返回的引用后面的指针。这是一个早已注定的资源泄漏。

但是也许你注意到无论是在栈上的还是在堆上的方法，为了从 operator* 返回的每一个 result，我们都不得不容忍一次构造函数的调用。也许你想起我们最初的目标是避免这样的构造函数调用。也许你认为你知道一种方法能避免除一次以外几乎全部的构造函数调用。也许下面这个实现是你做过的，一个基于 operator* 返回一个引向 static Rational 对象的引用的实现，而这个 static Rational 对象定义在函数内部：

```
const Rational& operator*(const Rational& lhs,    // warning! yet more
                          const Rational& rhs)    // bad code!
{
  static Rational result;             // static object to which a
                                      // reference will be returned

  result = ... ;                      // multiply lhs by rhs and put the
                                      // product inside result
  return result;
}
```

就像所有使用了 static 对象的设计一样，这个也会立即引起我们的线程安全（thread-safety）的混乱，但那是它的比较明显的缺点。为了看到它的更深层的缺陷，考虑这个完全合理的客户代码：

```
bool operator==(const Rational& lhs,            // an operator==
                const Rational& rhs);           // for Rationals

Rational a, b, c, d;

...
if ((a * b) == (c * d))  {
    do whatever's appropriate when the products are equal;
} else    {
    do whatever's appropriate when they're not;
}
```

猜猜会怎么样？不管 a，b，c，d 的值是什么，表达式 ((a*b*) == (*c*d)) 总是等于 true！

如果代码重写为功能完全等价的另一种形式，这一启示就很容易被理解了：

```
if (operator==(operator*(a, b), operator*(c, d)))
```

注意，当 operator== 被调用时，将同时存在两个起作用的对 operator* 的调用，每一个都将返回引向 operator* 内部的 static Rational 对象的引用。因此，operator== 将被要求比较 operator* 内部的 static Rational 对象的值和 operator* 内部的 static Rational 对象的值。如果它们不是永远相等，那才真的会令人大惊失色了。

这些应该足够让你信服试图从类似 operator* 这样的函数中返回一个引用纯粹是浪费时间，但是你们中的某些人可能会这样想“好吧，就算一个 static 不够用，也许一个 static 的数组是一个窍门……”

我无法拿出示例代码来肯定这个设计，但我可以概要说明为什么这个想法应该让你羞愧得无地自容。首先，你必须选择一个 n 作为数组的大小。如果 n 太小，你可能会用完存储函数返回值的空间，与刚刚名誉扫地的 single-static 设计相比，在任何一个方面你都不会得到更多的东西。但是如果 n 太大，就会降低你的程序的性能，因为在函数第一次被调用的时候数组中的每一个对象都会被构造。即使这个我们正在讨论的函数仅被调用了一次，也将让你付出 n 个构造函数和 n 个析构函数的成本。如果“优化”是提高软件效率的过程，对于这种东西也只能是“悲观主义”的。最后，考虑你怎样将你所需要的值放入数组的对象中，以及你做这些需要付出什么。在两个对象间移动值的最直接方法就是通过赋值，但是一次赋值将要付出什么？对于很多类型，这就大约相当于调用一次析构函数（销毁原来的值）加上调用一次构造函数（把新值拷贝过去）。但是你的目标是避免付出构造和析构成本！面对的结果就是：这个方法绝对不会成功。（不，用一个 vector 代替数组也不会让事情有多少改进。）

写一个必须返回一个新对象的函数的正确方法就是让那个函数返回一个新对象。对于 Rational 的 operator*，这就意味着下面这些代码或在本质上与其相当的某些东西：

```
inline const Rational operator*(const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

当然，你可能付出了构造和析构 operator* 的返回值的成本，但是从长远看，这只是为正确行为付出的很小的代价。除此之外，这种令你感到恐怖的账单也许永远都不会到达。就像所有的程序设计语言，C++ 允许编译器的实现者在不改变生成代码的可观察行为的条件下使用优化来提升它的性能，在某些条件下会产生如下结果：operator* 的返回值的构造和析构能被安全地消除。如果编译器利用了这一点（编译器经常这样做），你的程序还是在它假定的方法上继续运行，只是比你期待的要快。

全部的焦点在这里：如果需要在返回一个引用和返回一个对象之间做出决定，你的工作就是让那个选择能提供正确的行为。让你的编译器厂商去绞尽脑汁使那个选择尽可能地廉价。

Things to Remember

- 绝不要返回一个局部栈对象的指针或引用，绝不要返回一个被分配的堆对象的引用，如果存在需要一个以上这样的对象的可能性时，绝不要返回一个局部 static 对象的指针或引用。（Item 4 提供的一个返回一个局部 static 的设计的例子是合理的，至少在单线程的环境中是这样。）

Item 22: 将数据成员声明为 private

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

好了，先公布一下计划。首先，我们将看看为什么数据成员不应该声明为 public。然后，我们将看到所有反对 public 数据成员的理由同样适用于 protected 数据成员。这就导出了数据成员应该是 private 的结论，至此，我们就结束了。

那么，public 数据成员，为什么不呢？

我们从先从语法一致性开始（参见 Item 18）。如果数据成员不是 public 的，客户访问一个对象的唯一方法就是通过成员函数。如果在 public 接口中的每件东西都是一个函数，客户就不必绞尽脑汁试图记住当他们要访问一个类的成员时是否需要使用圆括号。他们只要使用就可以了，因为每件东西都是一个函数。一生坚持这一方针，能节省很多挠头的时间。

但是也许你不认为一致性的理由是强制性的。使用函数可以让你更加精确地控制成员的可存取性的事实又怎么样呢？如果你让一个数据成员为 public，每一个人都可以读写访问它，但是如果你使用函数去得到和设置它的值，你就能实现禁止访问，只读访问和读写访问。嘿嘿，如果你需要，你甚至可以实现只写访问：

```
class AccessLevels {
public:
  ...
  int getReadOnly() const        { return readOnly; }

  void setReadWrite(int value)   { readWrite = value; }
  int getReadWrite() const       { return readWrite; }

  void setWriteOnly(int value)   { writeOnly = value; }

private:
  int noAccess;                         // no access to this int

  int readOnly;                         // read-only access to this int

  int readWrite;                        // read-write access to this int

  int writeOnly;                        // write-only access to this int
};
```

这种条分缕析的访问控制很重要，因为多数数据成员需要被隐藏。每一个数据成员都需要一个 getter 和 setter 的情况是很罕见的。

还不相信吗？那么该拿出一门重炮了：封装。如果你通过一个函数实现对数据成员的访问，你可以在以后用一个计算来替换这个数据成员，使用你的类的人不会有任何察觉。

例如，假设你为一个监视通过的汽车的速度的自动设备写一个应用程序。每通过一辆汽车，它的速度就被计算，而且那个值要加入到迄今为止收集到的所有速度数据的集合中：

```
class SpeedDataCollection {

  ...
public:
  void addValue(int speed);          // add a new data value

  double averageSoFar() const;       // return average speed

  ...
};
```

现在考虑成员函数 averageSoFar 的实现：实现它的办法之一是在类中用一个数据成员来实时变化迄今为止收集到的所有速度数据的平均值。无论何时 averageSoFar 被调用，它只是返回那个数据成员的值。另一个不同的方法是在每次调用 averageSoFar 时重新计算它的值，通过分析集合中每一个数据值它能做成这些事情。

第一种方法（保持一个实时变化的值）使每一个 SpeedDataCollection 对象都比较大，因为你必须为持有实时变化的平均值，累计的和以及数据点的数量分配空间。可是，averageSoFar 能实现得非常高效，它仅仅是一个返回实时变化的平均值的 inline 函数（参见 Item 30）。反过来，无论何时被请求都要计算平均值使得 averageSoFar 的运行比较慢，但是每一个 SpeedDataCollection 对象都比较小。

谁能说哪一个最好？在内存非常紧张的机器（例如，一个嵌入式道旁设备）上，以及在一个很少需要平均值的应用程序中，每次都计算平均值可能是较好的解决方案。在一个频繁需要平均值的应用程序中，速度是基本的要求，而且内存不成问题，保持一个实时变化的平均值更为可取。这里的重点在于通过经由一个成员函数访问平均值（也就是说，通过将它封装），你能互换这两个不同的实现（也包括其他你可能想到的），对于客户，最多也就是必须重新编译。（你可以用在后面的 Item 31 中记述的技术来消除这个麻烦。）

将数据成员隐藏在功能性的接口之后能为各种实现提供弹性。例如，它可以在读或者写的时候很简单地通报其他对象，可以检验类的不变量以及函数的前置或后置条件，可以在多线程环境中执行同步任务，等等。从类似 Delphi 和 C# 的语言来到 C++ 的程序员会认同这种类似那些语言中的“属性”的等价物的功能，虽然需要附加一个带圆括号的额外的 set。

关于封装的要点可能比它最初显现出来的更加重要。如果你对你的客户隐藏你的数据成员（也就是说，封装它们），你就能确保类的不变量总能被维持，因为只有成员函数能影响它们。此外，你预留了以后改变你的实现决策的权力。如果你不隐藏这样的决策，你将很快发现，即使你拥有一个类的源代码，你改变任何一个 public 的东西的能力也是非常有限的，因为有太多的客户代码将被破坏。public 意味着没有封装，而且几乎可以说，没有封装意味着不可改变，尤其是被广泛使用的类。但是仍然被广泛使用的类大多数都是需要封装的，因为它们可以从用一种更好的实现替换现有实现的能力中获得最多的益处。

反对 protected 数据成员的理由是类似的。实际上，它是一样的，虽然起先看起来似乎不那么清楚。关于语法一致性和条分缕析的访问控制的论证就像用于 public 一样可以应用于 protected，但是关于封装又如何呢？难道 protected 数据成员不比 public 数据成员更具有封装性吗？实话实说，令人惊讶的答案是它们不。

Item 23 解释了如果某物发生了变化，某物的封装与可能被破坏的代码数量成反比。于是，如果数据成员发生了变化（例如，如果它被从类中移除（可能是为了替换为计算，就像在上面的 averageSoFar 中）），数据成员的封装性与可能被破坏的代码数量成反比。

假设我们有一个 public 数据成员，随后我们消除了它。有多少代码会被破坏呢？所有使用了它的客户代码，其数量通常大得难以置信。从而 public 数据成员就是完全未封装的。但是，假设我们有一个 protected 数据成员，随后我们消除了它。现在有多少代码会被破坏呢？所有使用了它的派生类，典型情况下，代码的数量还是大得难以置信。从而 protected 数据成员就像 public 数据成员一样没有封装，因为在这两种情况下，如果数据成员发生变化，被破坏的客户代码的数量都大得难以置信。这并不符合直觉，但是富有经验的库实现者会告诉你，这是千真万确的。一旦你声明一个数据成员为 public 或 protected，而且客户开始使用它，就很难再改变与这个数据成员有关的任何事情。有太多的代码不得不被重写，重测试，重文档化，或重编译。从封装的观点来看，实际只有两个访问层次：private（提供了封装）与所有例外（没有提供封装）。

Things to Remember

声明数据成员为 private。它为客户提供了访问数据的语法层上的一致，提供条分缕析的访问控制，允许不变量被强制，而且为类的作者提供了实现上的弹性。

protected 并不比 public 的封装性强。

Item 23: 用非成员非友元函数取代成员函数

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

想象一个象征 web 浏览器的类。在大量的函数中，这样一个类也许会提供清空已下载成分的缓存。清空已访问 URLs 的历史，以及从系统移除所有 cookies 的功能：

```
class WebBrowser {
public:
  ...
  void clearCache();
  void clearHistory();
  void removeCookies();
  ...
};
```

很多用户希望能一起执行全部这些动作，所以 WebBrowser 可能也会提供一个函数去这样做：

```
class WebBrowser {
public:
  ...
  void clearEverything();               // calls clearCache, clearHistory,
                                        // and removeCookies
  ...
};
```

当然，这个功能也能通过非成员函数调用适当的成员函数来提供：

```
void clearBrowser(WebBrowser& wb)
{
  wb.clearCache();
  wb.clearHistory();
  wb.removeCookies();
}
```

那么哪个更好呢，成员函数 clearEverything 还是非成员函数 clearBrowser？

面性对象原则指出：数据和对它们进行操作的函数应该被绑定到一起，而且建议成员函数是更好的选择。不幸的是，这个建议是不正确的。它产生于对面向对象是什么的一个误解。面向对象原则指出数据应该尽可能被封装。与直觉不同，成员函数 clearEverything 居然会造成比非成员函数 clearBrowser 更差的封装性。此外，提供非成员函数允许 WebBrowser 相关功能的更大的包装弹性，而且，可以获得更少的编译依赖和 WebBrowser 扩展性的增进。因而，在很多方面非成员方法比一个成员函数更好。理解它的原因是非常重要的。

我们将从封装开始。如果某物被封装，它被从视线中隐藏。越多的东西被封装，就越少有东西能看见它。越少有东西能看见它，我们改变它的弹性就越大，因为我们的改变仅仅直接影响那些能看见我们变了什么的东西。某物的封装性越强，那么我们改变它的能力就越强。这就是将封装的价值评价为第一的原因：它为我们提供一种改变事情的弹性，而仅仅影响有限的客户。

结合一个对象考虑数据。越少有代码能看到数据（也就是说，访问它），数据封装性就越强，我们改变对象的数据的特性的自由也就越大，比如，数据成员的数量，它们的类型，等等。作为多少代码能看到一块数据的粗糙的尺度，我们可以计数能访问那块数据的函数的数量：越多函数能访问它，数据的封装性就越弱。

Item 22 说明了数据成员应该是 private 的，因为如果它们不是，就有无限量的函数能访问它们。它们根本就没有封装。对于 private 数据成员，能访问他们的函数的数量就是类的成员函数的数量加上友元函数的数量，因为只有成员和友元能访问 private 成员。假设在一个成员函数（能访问的不只是一个类的 private 数据，还有 private 函数，枚举，typedefs，等等）和一个提供同样功能的非成员非友元函数（不能访问上述那些东西）之间有一个选择，能获得更强封装性的选择是非成员非友元函数，因为它不会增加能访问类的 private 部分的函数的数量。这就解释了为什么 clearBrowser（非成员非友元函数）比 clearEverything（成员函数）更可取：它能为 WebBrowser 获得更强的封装性。

在这一点，有两件事值得注意。首先，这个论证只适用于非成员非友元函数。友元能像成员函数一样访问一个类的 private 成员，因此同样影响封装。从封装的观点看，选择不是在成员和非成员函数之间，而是在成员函数和非成员非友元函数之间。（当然，封装并不是仅有的观点，Item 24 说明如果观点来自隐式类型转换，选择就是在成员和非成员函数之间。）

需要注意的第二件事是，如果仅仅是为了关注封装，则可以指出，一个函数是一个类的非成员并不意味着它不可以是另一个类的成员。这对于习惯了所有函数必须属于类的语言（例如，Eiffel，Java，C#，等等）的程序员是一个适度的安慰。例如，我们可以使 clearBrowser 成为一个 utility 类的 static 成员函数。只要它不是 WebBrowser 的一部分（或友元），它就不会影响 WebBrowser 的 private 成员的封装。

在 C++ 中，一个更自然的方法是使 clearBrowser 成为与 WebBrowser 在同一个 namespace（名字空间）中的非成员函数：

```
namespace WebBrowserStuff {

 class WebBrowser { ... };

 void clearBrowser(WebBrowser& wb);

  ...
}
```

相对于形式上的自然，这样更适用于它。无论如何，因为名字空间（不像类）能展开到多个源文件中。这是很重要的，因为类似 clearBrowser 的函数是方便性函数。作为既不是成员也不是友元，他们没有对 WebBrowser 进行专门的访问，所以他们不能提供任何一种

WebBrowser 的客户不能通过其它方法得到的功能。例如，如果 clearBrowser 不存在，客户可以直接调用 clearCache，clearHistory 和 removeCookies 本身。

一个类似 WebBrowser 的类可以有大量的方便性函数，一些是书签相关的，另一些打印相关的，还有一些是 cookie 管理相关的，等等。作为一个一般的惯例，多数客户仅对这些方便性函数的集合中的一些感兴趣。没有理由让一个只对书签相关的方便性函数感兴趣的客户在编译时依赖其它函数，例如，cookie 相关的方便性函数。分隔它们的直截了当的方法就是在一个头文件中声明书签相关的方便性函数，在另一个不同的头文件中声明 cookie 相关的方便性函数，在第三个头文件声明打印相关的方便性函数，等等：

```
// header "webbrowser.h" — header for class WebBrowser itself
// as well as "core" WebBrowser-related functionality
namespace WebBrowserStuff {

   class WebBrowser { ... };

     ...                                // "core" related functionality, e.g.
                                        // non-member functions almost
                                        // all clients need
}
// header "webbrowserbookmarks.h"
namespace WebBrowserStuff {
  ...                                   // bookmark-related convenience
}                                       // functions
// header "webbrowsercookies.h"
namespace WebBrowserStuff {
  ...                                   // cookie-related convenience
}                                       // functions

...
```

注意这里就像标准 C++ 库组织得一样严密。胜于有一个单独的一体式的 <C++StandardLibrary> 头文件包含 std namespace 中的所有东西，它们在许多头文件中（例如，<vector>，<algorithm>，<memory>，等等），每一个都声明了 std 中的一些机能。仅仅需要 vector 相关机能的客户不需要 #include <memory>，不用 list 的客户没有必要 #include <list>。这就允许客户在编译时仅仅依赖他们实际使用的那部分系统。（参见 Item 31 对减少编译依赖的其它方法的讨论。）当机能来自一个类的成员函数时，用这种方法分割它是不可能的，因为一个类必须作为一个整体来定义，它不能四分五裂。

将所有方便性函数放入多个头文件中——但是在一个 namespace 中——也意味着客户能容易地扩充方便性函数的集合。他们必须做的全部就是在 namespace 中加入更多的非成员非友元函数。例如，如果一个 WebBrowser 的客户决定写一个关于下载图像的方便性函数，他或她仅仅需要新建一个头文件，包含那些函数在 WebBrowserStuff namespace 中的声明。这个新的函数现在就像其它方便性函数一样可用并被集成。这是类不能提供的另一个特性，因为类定义对于客户是扩充封闭的。当然，客户可以派生新类，但是派生类不能访问基类中被封装的（也就是说，private 的）成员，所以这样的“扩充机能”只有二等身份。此外，就像 Item 7 中解释的，不是所有的类都是作为基类设计的。

Things to Remember

- 用非成员非友元函数取代成员函数。这样做可以提高封装性，包装弹性，和机能扩充性。

Item 24: 当类型转换应该用于所有参数时，声明为非成员函数

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在此书的 Introduction 中我谈到让一个类支持隐式类型转换通常是一个不好的主意。当然，这条规则有一些例外，最普通的一种就是在创建数值类型时。例如，如果你设计一个用来表现有理数的类，允许从整数到有理数的隐式转换看上去并非不合理。这的确不比 C++ 的内建类型从 int 到 double 的转换更不合理（而且比 C++ 的内建类型从 double 到 int 的转换合理得多）。在这种情况下，你可以用这种方法开始你的 Rational 类：

```
class Rational {
public:
  Rational(int numerator = 0,        // ctor is deliberately not explicit;
           int denominator = 1);     // allows implicit int-to-Rational
                                     // conversions

  int numerator() const;             // accessors for numerator and
  int denominator() const;           // denominator – see Item 22

private:
  ...
};
```

你知道你应该支持算术运算，比如加法，乘法，等等，但是你不能确定是通过成员函数，非成员函数，还是非成员的友元函数来实现它们。你的直觉告诉你，当你摇摆不定的时候，你应该坚持面向对象的原则。你了解这一点，于是断定，因为有理数的乘法与 Rational 类相关，所以在 Rational 类的内部实现有理数的 operator* 似乎更加正常。但是，与直觉不符的是，Item 23 指出将函数放在它们所关联的类的内部的主张有时候与面向对象的原则正好相反，但是让我们将它先放到一边，来研究一下让 operator* 成为 Rational 的一个成员函数的想法究竟如何：

```
class Rational {
public:
 ...

 const Rational operator*(const Rational& rhs) const;
};
```

（如果你不能确定为什么这个函数声明为这个样子——返回一个 const by-value 的结果，却持有一个 reference-to-const 作为它的参数——请参考 Item 3，20 和 21。）

这个设计让你在有理数相乘时不费吹灰之力：

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;            // fine

result = result * oneEighth;                      // fine
```

但是你并不感到满意。你还希望支持混合模式的操作，以便让 Rationals 能够和其它类型（例如，int）相乘。毕竟，很少有事情像两个数相乘那么正常，即使它们碰巧是数字的不同类型。

当你试图做混合模式的算术运算时，可是，你发现只有一半时间它能工作：

```
result = oneHalf * 2; // fine

result = 2 * oneHalf; // error!
```

这是一个不好的征兆。乘法必须是可交换的，记得吗？

当你重写最后两个例子为功能等价的另一种形式时，问题的来源就变得很明显了：

```
result = oneHalf.operator*(2); // fine

result = 2.operator*(oneHalf); // error!
```

对象 oneHalf 是一个包含 operator* 的类的实例，所以编译器调用那个函数。然而，整数 2 与类没有关系，因而没有 operator* 成员函数。编译器同样要寻找能如下调用的非成员的 operator*s（也就是说，在 namespace 或全局范围内的 operator*s）：

```
result = operator*(2, oneHalf); // error!
```

但是在本例中，没有非成员的持有一个 int 和一个 Rational 的 operator*，所以搜索失败。

再看一眼那个成功的调用。你会发现它的第二个参数是整数 2，然而 Rational::operator* 却持有一个 Rational 对象作为它的参数。这里发生了什么呢？为什么 2 在一个位置能工作，在其它地方却不行呢？

发生的是隐式类型转换。编译器知道你传递一个 int 而那个函数需要一个 Rational，但是它们也知道通过用你提供的 int 调用 Rational 的构造函数，它们能做出一个相配的 Rational，这就是它们的所作所为。换句话说，它们将那个调用或多或少看成如下这样：

```
const Rational temp(2);              // create a temporary
                                     // Rational object from 2

result = oneHalf * temp;             // same as oneHalf.operator*(temp);
```

当然，编译器这样做仅仅是因为提供了一个非显性的构造函数。如果 Rational 的构造函数是显性的，这些语句都将无法编译：

```
result = oneHalf * 2;                // error! (with explicit ctor);
                                     // can't convert 2 to Rational

result = 2 * oneHalf;                // same error, same problem
```

支持混合模式操作失败了，但是至少两个语句的行为将步调一致。

然而，你的目标是既保持一致性又要支持混合运算，也就是说，一个能使上面两个语句都可以编译的设计。让我们返回这两个语句看一看，为什么即使 Rational 的构造函数不是显式的，也是一个可以编译而另一个不行：

```
result = oneHalf * 2; // fine (with non-explicit ctor)

result = 2 * oneHalf; // error! (even with non-explicit ctor)
```

其原因在于仅仅当参数列在参数列表中的时候，它们才有资格进行隐式类型转换。而对应于成员函数被调用的那个对象的隐含参数—— this 指针指向的那个——根本没有资格进行隐式转换。这就是为什么第一个调用能编译而第二个不能。第一种情况包括一个参数被列在参数列表中，而第二种情况没有。

你还是希望支持混合运算，然而，现在做到这一点的方法或许很清楚了：让 operator* 作为非成员函数，因此就允许便一起将隐式类型转换应用于所有参数：

```
class Rational {

  ...                                             // contains no operator*
};
const Rational operator*(const Rational& lhs,     // now a non-member
                         const Rational& rhs)     // function
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;                           // fine
result = 2 * oneFourth;                           // hooray, it works!
```

这样的确使故事有了一个圆满的结局，但是有一个吹毛求疵的毛病。operator* 应该不应该作为 Rational 类的友元呢？

在这种情况下，答案是不，因为 operator* 能够根据 Rational 的 public 接口完全实现。上面的代码展示了做这件事的方法之一。这导出了一条重要的结论：与成员函数相对的是非成员函数，而不是友元函数。太多的程序员假设如果一个函数与一个类有关而又不应该作为成员时（例如，因为所有的参数都需要类型转换），它应该作为友元。这个示例证明这样的推理

是有缺陷的。无论何时，只有你能避免友元函数，你就避免它，因为，就像在现实生活中，朋友的麻烦通常多于他们的价值。当然，有时友谊是正当的，但是事实表明仅仅因为函数不应该作为成员并不自动意味着它应该作为友元。

本 Item 包含真理，除了真理一无所有，但它还不是完整的真理。当你从 Object-Oriented C++ 穿过界线进入 Template C++（参见 Item 1）而且将 Rational 做成一个类模板代替一个类，就有新的问题要考虑，也有新的方法来解决它们，以及一些令人惊讶的设计含义。这样的问题，解决方法和含义是 Item 46 的主题。

Things to Remember

- 如果你需要在一个函数的所有参数（包括被 this 指针所指向的那个）上使用类型转换，这个函数必须是一个非成员。

Item 25: 考虑支持不抛异常的 swap

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

swap 是一个有趣的函数。最早作为 STL 的一部分被引入，后来它成为异常安全编程（exception-safe programming）的支柱（参见 Item 29）和压制自赋值可能性的通用机制（参见 Item 11）。因为 swap 太有用了，所以正确地实现它非常重要，但是伴随它的不同寻常的重要性而来的，是一系列不同寻常的复杂性。在本 Item 中，我们就来研究一下这些复杂性究竟是什么样的以及如何对付它们。

交换两个对象的值就是互相把自己的值送给对方。缺省情况下，通过标准的交换算法来实现交换是非常成熟的技术。典型的实现完全符合你的预期：

```
namespace std {

  template<typename T>          // typical implementation of std::swap;
  void swap(T& a, T& b)         // swaps a's and b's values
  {
    T temp(a);
    a = b;
    b = temp;
  }
}
```

只要你的类型支持拷贝（通过拷贝构造函数和拷贝赋值运算符），缺省的 swap 实现就能交换你的类型的对象，而不需要你做任何特别的支持工作

可是，缺省的 swap 实现可能不那么酷。它涉及三个对象的拷贝：从 a 到 temp，从 b 到 a，以及从 temp 到 b。对一些类型来说，这些副本全是不必要的。对于这样的类型，缺省的 swap 就好像让你坐着快车驶入小巷。

这样的类型中最重要的就是那些主要由一个指针组成的类型，那个指针指向包含真正数据的另一种类型。这种设计方法的一种常见的表现形式是 "pimpl idiom"（"pointer to implementation" ——参见 Item 31）。一个使用了这种设计的 Widget 类可能就像这样：

```
class WidgetImpl {                          // class for Widget data;
public:                                     // details are unimportant
  ...

private:
  int a, b, c;                              // possibly lots of data —
  std::vector<double> v;                    // expensive to copy!
  ...
};

class Widget {                              // class using the pimpl idiom
public:
  Widget(const Widget& rhs);

  Widget& operator=(const Widget& rhs)      // to copy a Widget, copy its
  {                                         // WidgetImpl object. For
   ...                                      // details on implementing
   *pImpl = *(rhs.pImpl);                   // operator= in general,
   ...                                      // see Items 10, 11, and 12.
  }
  ...

private:
  WidgetImpl *pImpl;                         // ptr to object with this
};
```

为了交换这两个 Widget 对象的值，我们实际要做的就是交换它们的 pImpl 指针，但是缺省的交换算法没有办法知道这些。它不仅要拷贝三个 Widgets，而且还有三个 WidgetImpl 对象，效率太低了。一点都不酷。

当交换 Widgets 的是时候，我们应该告诉 std::swap 我们打算做什么，执行交换的方法就是交换它们内部的 pImpl 指针。这种方法的正规说法是：针对 Widget 特化 std::swap（specialize std::swap for Widget）。下面是一个基本的想法，虽然在这种形式下它还不能通过编译：

```
namespace std {

  template<>                            // this is a specialized version
  void swap<Widget>(Widget& a,          // of std::swap for when T is
                    Widget& b)          // Widget; this won't compile
  {
    swap(a.pImpl, b.pImpl);             // to swap Widgets, just swap
  }                                     // their pImpl pointers
}
```

这个函数开头的 "template<>" 表明这是一个针对 std::swap 的完全模板特化（total template specialization）（某些书中称为 "full template specialization" 或 "complete template specialization" ——译者注），函数名后面的 "<Widget>" 表明特化是在 T 为 Widget 类型时发生的。换句话说，当通用的 swap 模板用于 Widgets 时，就应该使用这个实现。通常，我们改变 std namespace 中的内容是不被允许的，但允许为我们自己创建的类型（就像 Widget）完全特化标准模板（就像 swap）。这就是我们现在在这里做的事情。

可是，就像我说的，这个函数还不能编译。那是因为它试图访问 a 和 b 内部的 pImpl 指针，而它们是 private 的。我们可以将我们的特化声明为友元，但是惯例是不同的：让 Widget 声明一个名为 swap 的 public 成员函数去做实际的交换，然后特化 std::swap 去调用那个成员函

数：

```
class Widget {                    // same as above, except for the
public:                           // addition of the swap mem func
  ...
  void swap(Widget& other)
  {
    using std::swap;              // the need for this declaration
                                  // is explained later in this Item

    swap(pImpl, other.pImpl);     // to swap Widgets, swap their
  }                               // pImpl pointers
  ...
};

namespace std {

  template<>                      // revised specialization of
  void swap<Widget>(Widget& a,    // std::swap
                    Widget& b)
  {
    a.swap(b);                    // to swap Widgets, call their
  }                               // swap member function
}
```

这个不仅能够编译，而且和 STL 容器保持一致，所有 STL 容器都既提供了 public swap 成员函数，又提供了 std::swap 的特化来调用这些成员函数。

可是，假设 Widget 和 WidgetImpl 是类模板，而不是类，或许因此我们可以参数化存储在 WidgetImpl 中的数据类型：

```
template<typename T>
class WidgetImpl { ... };

template<typename T>
class Widget { ... };
```

在 Widget 中加入一个 swap 成员函数（如果我们需要，在 WidgetImpl 中也加一个）就像以前一样容易，但我们特化 std::swap 时会遇到麻烦。这就是我们要写的代码：

```
namespace std {
  template<typename T>
  void swap<Widget<T> >(Widget<T>& a,      // error! illegal code!
                        Widget<T>& b)
  { a.swap(b); }
}
```

这看上去非常合理，但它是非法的。我们试图部分特化（partially specialize）一个函数模板（std::swap），但是尽管 C++ 允许类模板的部分特化（partial specialization），但不允许函数模板这样做。这样的代码不能编译（尽管一些编译器错误地接受了它）。

当我们想要“部分特化”一个函数模板时，通常做法是简单地增加一个重载。看起来就像这样：

```
namespace std {

  template<typename T>              // an overloading of std::swap
  void swap(Widget<T>& a,           // (note the lack of "<...>" after
            Widget<T>& b)           // "swap"), but see below for
  { a.swap(b); }                    // why this isn't valid code
}
```

通常，重载函数模板确实很不错，但是 std 是一个特殊的 namespace，规则对它也有特殊的待遇。它认可完全特化 std 中的模板，但它不认可在 std 中增加新的模板（也包括类，函数，以及其它任何东西）。std 的内容由 C++ 标准化委员会单独决定，并禁止我们对他们做出的决定进行增加。而且，禁止的方式使你无计可施。打破这条禁令的程序差不多的确可以编译和运行，但它们的行为是未定义的。如果你希望你的软件有可预期的行为，你就不应该向 std 中加入新的东西。

因此该怎么做呢？我们还是需要一个方法，既使其他人能调用 swap，又能让我们得到更高效的模板特化版本。答案很简单。我们还是声明一个非成员 swap 来调用成员 swap，只是不再将那个非成员函数声明为 std::swap 的特化或重载。例如，如果我们的 Widget 相关机能都在 namespace WidgetStuff 中，它看起来就像这个样子：

```
namespace WidgetStuff {
  ...                                      // templatized WidgetImpl, etc.

  template<typename T>                     // as before, including the swap
  class Widget { ... };                    // member function

  ...

  template<typename T>                     // non-member swap function;
  void swap(Widget<T>& a,                  // not part of the std namespace
            Widget<T>& b)
  {
    a.swap(b);
  }
}
```

现在，如果某处有代码使用两个 Widget 对象调用 swap，C++ 的名字查找规则（以参数依赖查找（argument-dependent lookup）或 Koenig 查找（Koenig lookup）著称的特定规则）将找到 WidgetStuff 中的 Widget 专用版本。而这正是我们想要的。

这个方法无论对于类模板还是对于类都能很好地工作，所以看起来我们应该总是使用它。不幸的是，此处还是存在一个需要为类特化 std::swap 的动机（过一会儿我会讲到它），所以如果你希望你的 swap 的类专用版本在尽可能多的上下文中都能够调用（而你也确实这样做了），你就既要在你的类所在的 namespace 中写一个非成员版本，又要提供一个 std::swap 的特化版本。

顺便提一下，如果你不使用 namespaces，上面所讲的一切依然适用（也就是说，你还是需要一个非成员 swap 来调用成员 swap），但是你为什么要把你的类，模板，函数，枚举（此处作者连用了两个词（enum, enumerant），不知有何区别——译者注）和 typedef 名字都堆在全局 namespace 中呢？你觉得合适吗？

迄今为止我所写的每一件事情都适用于 swap 的作成者，但是有一种状况值得从客户的观点来看一看。假设你写了一个函数模板来交换两个对象的值：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  ...
  swap(obj1, obj2);
  ...
}
```

哪一个 swap 应该被调用呢？std 中的通用版本，你知道它必定存在；std 中的通用版本的特化，可能存在，也可能不存在；T 专用版本，可能存在，也可能不存在，可能在一个 namespace 中，也可能不在一个 namespace 中（但是肯定不在 std 中）。究竟该调用哪一个呢？如果 T 专用版本存在，你希望调用它，如果它不存在，就回过头来调用 std 中的通用版本。如下这样就可以符合你的希望：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
  using std::swap;           // make std::swap available in this function
  ...
  swap(obj1, obj2);          // call the best swap for objects of type T
  ...
}
```

当编译器看到这个 swap 调用，他会寻找正确的 swap 版本来调用。C++ 的名字查找规则确保能找到在全局 namespace 或者与 T 同一个 namespace 中的 T 专用的 swap。（例如，如果 T 是 namespace WidgetStuff 中的 Widget，编译器会利用参数依赖查找（argument-dependent lookup）找到 WidgetStuff 中的 swap。）如果 T 专用 swap 不存在，编译器将使用 std 中的 swap，这归功于此函数中的 using declaration 使 std::swap 在此可见。尽管如此，相对于通用模板，编译器还是更喜欢 T 专用的 std::swap 的特化，所以如果 std::swap 对 T 进行了特化，则特化的版本会被使用。

得到正确的 swap 调用是如此地容易。你需要小心的一件事是不要对调用加以限定，因为这将影响 C++ 确定该调用的函数，如果你这样写对 swap 的调用，

```
std::swap(obj1, obj2); // the wrong way to call swap
```

这将强制编译器只考虑 std 中的 swap（包括任何模板特化），因此排除了定义在别处的更为适用的 T 专用版本被调用的可能性。唉，一些被误导的程序员就是用这种方法限定对 swap 的调用，这也就是为你的类完全地特化 std::swap 很重要的原因：它使得以这种被误导的方式写出的代码可以用到类型专用的 swap 实现。（这样的代码还存在于现在的一些标准库实现中，所以它将有利于你帮助这样的代码尽可能高效地工作。）

到此为止，我们讨论了缺省的 swap，成员 swaps，非成员 swaps，std::swap 的特化版本，以及对 swap 的调用，所以让我们总结一下目前的状况。

首先，如果 swap 的缺省实现为你的类或类模板提供了可接受的性能，你不需要做任何事。任何试图交换你的类型的对象的人都会得到缺省版本的支持，而且能工作得很好。

第二，如果 swap 的缺省实现效率不足（这几乎总是意味着你的类或模板使用了某种 pimpl idiom 的变种），就按照以下步骤来做：

1. 提供一个能高效地交换你的类型的两个对象的值的 public 的 swap 成员函数。出于我过一会儿就要解释的动机，这个函数应该永远不会抛出异常。

2. 在你的类或模板所在的同一个 namespace 中提供一个非成员的 swap。用它调用你的 swap 成员函数。

3. 如果你写了一个类（不是类模板），就为你的类特化 std::swap。用它也调用你的 swap 成员函数。

最后，如果你调用 swap，请确保在你的函数中包含一个 using declaration 使 std::swap 可见，然后在调用 swap 时不使用任何 namespace 限定条件。

唯一没有解决的问题就是我的警告——绝不要让 swap 的成员版本抛出异常。这是因为 swap 的非常重要的应用之一是为类（以及类模板）提供强大的异常安全（exception-safety）保证。Item 29 将提供所有的细节，但是这项技术基于 swap 的成员版本绝不会抛出异常的假设。这一强制约束仅仅应用在成员版本上！它不能够应用在非成员版本上，因为 swap 的缺省版本基于拷贝构造和拷贝赋值，而在通常情况下，这两个函数都允许抛出异常。如果你写了一个 swap 的自定义版本，那么，典型情况下你是为了提供一个更有效率的交换值的方法，你也要保证这个方法不会抛出异常。作为一个一般规则，这两种 swap 的特型将紧密地结合在一起，因为高效的交换几乎总是基于内建类型（诸如在 pimpl idiom 之下的指针）的操作，而对内建类型的操作绝不会抛出异常。

Things to Remember

- 如果 std::swap 对于你的类型来说是低效的，请提供一个 swap 成员函数。并确保你的 swap 不会抛出异常。
- 如果你提供一个成员 swap，请同时提供一个调用成员 swap 的非成员 swap。对于类（非模板），还要特化 std::swap。
- 调用 swap 时，请为 std::swap 使用一个 using declaration，然后在调用 swap 时不使用任何 namespace 限定条件。
- 为用户定义类型完全地特化 std 模板没有什么问题，但是绝不要试图往 std 中加入任何全新的东西。

Item 26: 只要有可能就推迟变量定义

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

只要你定义了一个带有构造函数和析构函数的类型的变量，当控制流程到达变量定义的时候会使你担负构造成本，而当变量离开作用域的时候会使你担负析构成本。如果有无用变量造成这一成本，你就要尽你所能去避免它。

你可能认为你从来不会定义无用的变量，但是也许你应该再想一想。考虑下面这个函数，只要 password 的长度满足要求，它就返回一个 password 的加密版本。如果 password 太短，函数就会抛出一个定义在标准 C++ 库中的 logic_error 类型的异常（参见 Item 54）：

```
// this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  string encrypted;

  if (password.length() < MinimumPasswordLength) {
      throw logic_error("Password is too short");
  }
  ...                        // do whatever is necessary to place an
                             // encrypted version of password in encrypted
  return encrypted;
}
```

对象 encrypted 在这个函数中并不是完全无用，但是如果抛出了一个异常，它就是无用的。换句话说，即使 encryptPassword 抛出一个异常，你也要为构造和析构 encrypted 付出代价。因此得出以下结论：你最好将 encrypted 的定义推迟到你确信你真的需要它的时候：

```
// this function postpones encrypted's definition until it's truly necessary
std::string encryptPassword(const std::string& password)
{
  using namespace std;

  if (password.length() < MinimumPasswordLength) {
     throw logic_error("Password is too short");
  }

  string encrypted;

  ...                      // do whatever is necessary to place an
                           // encrypted version of password in encrypted
  return encrypted;
}
```

这一代码仍然没有达到它本可以达到的那样紧凑，因为定义 encrypted 的时候没有任何初始化参数。这就意味着很多情况下将使用它的缺省构造函数，对于一个对象你首先应该做的就是给它一些值，这经常可以通过赋值来完成。Item 4 解释了为什么缺省构造（default-constructing）一个对象然后赋值给它比用你真正需要它持有的值初始化它更低效。那个分析也适用于此。例如，假设 encryptPassword 的核心部分是在这个函数中完成的：

```
void encrypt(std::string& s); // encrypts s in place
```

那么，encryptPassword 就可以这样实现，即使它还不是最好的方法：

```
// this function postpones encrypted's definition until
// it's necessary, but it's still needlessly inefficient
std::string encryptPassword(const std::string& password)
{
  ...                                   // check length as above

  string encrypted;                     // default-construct encrypted
  encrypted = password;                 // assign to encrypted

  encrypt(encrypted);
  return encrypted;
}
```

一个更可取得方法是用 password 初始化 encrypted，从而跳过毫无意义并可能很昂贵的缺省构造：

```
// finally, the best way to define and initialize encrypted
std::string encryptPassword(const std::string& password)
{
  ...                                     // check length

  string encrypted(password);             // define and initialize
                                          // via copy constructor

  encrypt(encrypted);
  return encrypted;
}
```

这个建议就是本 Item 的标题中的“只要有可能（as long as possible）”的真正含义。你不仅应该推迟一个变量的定义直到你不得不用它之前的最后一刻，而且应该试图推迟它的定义直到你得到了它的初始化参数。通过这样的做法，你可以避免构造和析构无用对象，而且还可以避免不必要的缺省构造。更进一步，通过在它们的含义已经非常明确的上下文中初始化它们，有助于对变量的作用文档化。

“但是对于循环会如何？”你可能会有这样的疑问。如果一个变量仅仅在一个循环内使用，是循环外面定义它并在每次循环迭代时赋值给它更好一些，还是在循环内部定义这个变量更好一些呢？也就是说，下面这两个大致的结构中哪个更好一些？

```
// Approach A: define outside loop   // Approach B: define inside loop

Widget w;
for (int i = 0; i < n; ++i){         for (int i = 0; i < n; ++i) {
  w = some value dependent on i;       Widget w(some value dependent on i);
  ...                                  ...
}
```

这里我将一个类型 string 的对象换成了一个类型 Widget 的对象，以避免对这个对象的构造、析构或赋值操作的成本的任何已有的预见。

对于 Widget 的操作而言，就是下面这两个方法的成本：

- 方法 A：1 个构造函数 + 1 个析构函数 + n 个赋值。
- 方法 B：n 个构造函数 + n 个析构函数。

对于那些赋值的成本低于一个构造函数/析构函数对的成本的类，方法 A 通常更高效。特别是在 n 变得很大的情况下。否则，方法 B 可能更好一些。此外，方法 A 与方法 B 相比，使得名字 w 在一个较大的区域（包含循环的那个区域）内均可见，这可能会破坏程序的易理解性和可维护性。因此得出以下结论：除非你确信以下两点：（1）赋值比构造函数/析构函数对成本更低，而且（2）你正在涉及你的代码中的性能敏感的部分，否则，你应该默认使用方法 B。

Things to Remember

- 只要有可能就推迟变量定义。这样可以增加程序的清晰度并提高程序的性能。

Item 27: 将强制转型减到最少

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 的规则设计为保证不会发生类型错误。在理论上，如果你的程序想顺利地通过编译，你就不应该试图对任何对象做任何不安全的或无意义的操作。这是一个非常有价值的保证，你不应该轻易地放弃它。

不幸的是，强制转型破坏了类型系统。它会引起各种各样的麻烦，其中一些容易被察觉，另一些则格外地微妙。如果你从 C，Java，或 C# 转到 C++，请一定注意，因为强制转型在那些语言中比在 C++ 中更有必要，危险也更少。但是 C++ 不是 C，也不是 Java，也不是 C#。在这一语言中，强制转型是一个你必须全神贯注才可以靠近的特性。

我们就从回顾强制转型的语法开始，因为对于同样的强制转型通常有三种不同的写法。C 风格（C-style）强制转型如下：

```
(T) expression // cast expression to be of type T
```

函数风格（Function-style）强制转型使用这样的语法：

```
T(expression) // cast expression to be of type T
```

这两种形式之间没有本质上的不同，它纯粹就是一个把括号放在哪的问题。我把这两种形式称为旧风格（old-style）的强制转型。

C++ 同时提供了四种新的强制转型形式（通常称为新风格的或 C++ 风格的强制转型）：

```
const_cast<T>(expression)
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
static_cast<T>(expression)
```

每一种适用于特定的目的：

- const_cast 一般用于强制消除对象的常量性。它是唯一能做到这一点的 C++ 风格的强制转型。
- dynamic_cast 主要用于执行“安全的向下转型（safe downcasting）”，也就是说，要确定一个对象是否是一个继承体系中的一个特定类型。它是唯一不能用旧风格语法执行的强制转型。也是唯一可能有重大运行时代价的强制转型。（过一会儿我再提供细节。）

- reinterpret_cast 是特意用于底层的强制转型，导致实现依赖（implementation-dependent）（就是说，不可移植）的结果，例如，将一个指针转型为一个整数。这样的强制转型在底层代码以外应该极为罕见。在本书中我只用了一次，而且还仅仅是在讨论你应该如何为裸内存（raw memory）写一个调谐分配者（debugging allocator）的时候（参见 Item 50）。

- static_cast 可以被用于强制隐型转换（例如，non-const 对象转型为 const 对象（就像 Item 3 中的），int 转型为 double，等等）。它还可以用于很多这样的转换的反向转换（例如，void* 指针转型为有类型指针，基类指针转型为派生类指针），但是它不能将一个 const 对象转型为 non-const 对象。（只有 const_cast 能做到。）

旧风格的强制转型依然合法，但是新的形式更可取。首先，在代码中它们更容易识别（无论是人还是像 grep 这样的工具都是如此），这样就简化了在代码中寻找类型系统被破坏的地方的过程。第二，更精确地指定每一个强制转型的目的，使得编译器诊断使用错误成为可能。例如，如果你试图使用一个 const_cast 以外的新风格强制转型来消除常量性，你的代码将无法编译。

当我要调用一个 explicit 构造函数用来传递一个对象给一个函数的时候，大概就是我仅有的使用旧风格的强制转换的时候。例如：

```
class Widget {
public:
  explicit Widget(int size);
  ...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                    // create Widget from int
                                           // with function-style cast

doSomeWork(static_cast<Widget>(15));       // create Widget from int
                                           // with C++-style cast
```

由于某种原因，有条不紊的对象创建感觉上不像一个强制转型，所以在这个强制转型中我多半会用函数风格的强制转型代替 static_cast。反过来说，在你写出那些导致 core dump 的代码时，你通常都感觉你有合理的理由，所以你最好忽略你的感觉并始终都使用新风格的强制转型。

很多程序员认为强制转型除了告诉编译器将一种类型看作另一种之外什么都没做，但这是错误的。任何种类的类型转换（无论是通过强制转型的显式的还是编译器添加的隐式的）都会导致运行时的可执行代码。例如，在这个代码片断中，

```
int x, y;
...
double d = static_cast<double>(x)/y;           // divide x by y, but use
                                               // floating point division
```

int x 到 double 的强制转型理所当然要生成代码，因为在大多数系统架构中，一个 int 的底层表示与 double 的不同。这可能还不怎么令人吃惊，但是下面这个例子可能会让你稍微开一下眼：

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                         // implicitly convert Derived* → Base*
```

这里我们只是创建了一个指向派生类对象的基类指针，但是有时候，这两个指针的值并不相同。在当前情况下，会在运行时在 Derived* 指针上应用一个偏移量以得到正确的 Base* 指针值。

这后一个例子表明一个单一的对象（例如，一个类型为 Derived 的对象）可能会有不止一个地址（例如，它的被一个 Base* 指针指向的地址和它的被一个 Derived* 指针指向的地址）。这在 C 中就不会发生，也不会在 Java 中发生，也不会在 C# 中发生，它仅在 C++ 中发生。实际上，如果使用了多继承，则一定会发生，但是在单继承下也会发生。与其它事情合在一起，就意味着你应该总是避免对 C++ 如何摆放事物做出假设，你当然也不应该基于这样的假设执行强制转型。例如，将一个对象的地址强制转型为 char* 指针，然后对其使用指针运算，这几乎总是会导致未定义行为。

但是请注意我说一个偏移量是“有时”被需要。对象摆放的方法和他们的地址的计算方法在不同的编译器之间有所变化。这就意味着仅仅因为你的“我知道事物是如何摆放的”而使得强制转型能工作在一个平台上，并不意味着它们也能在其它平台工作。这个世界被通过痛苦的道路学得这条经验的可怜的程序员所充满。

关于强制转型的一件有趣的事是很容易写出看起来对（在其它语言中也许是对的）实际上错的东西。例如，许多应用框架（application framework）要求在派生类中实现虚成员函数时要首先调用它们的基类对应物。假设我们有一个 Window 基类和一个 SpecialWindow 派生类，它们都定义了虚函数 onResize。进一步假设 SpecialWindow 的 onResize 被期望首先调用 Window 的 onResize。这就是实现这个的一种方法，它看起来正确实际并不正确：

```
class Window {                                // base class
public:
  virtual void onResize() { ... }             // base onResize impl
  ...
};

class SpecialWindow: public Window {          // derived class
public:
  virtual void onResize() {                   // derived onResize impl;
    static_cast<Window>(*this).onResize();    // cast *this to Window,
                                              // then call its onResize;
                                              // this doesn't work!

    ...                                       // do SpecialWindow-
  }                                           // specific stuff

  ...

};
```

我突出了代码中的强制转型。（这是一个新风格的强制转型，但是使用旧风格的强制转型也于事无补。）正像你所期望的，代码将 *this* 强制转型为一个 *Window*。因此调用 *onResize* 的结果就是调用 *Window::onResize*。你也许并不期待它没有调用当前对象的那个函数！作为替代，强制转型创建了一个 this 的基类部分的新的，临时的拷贝，然后调用这个拷贝的 onResize！上面的代码没有调用当前对象的 Window::onResize，然后再对这个对象执行 SpecialWindow 特有的动作——它在对当前对象执行 SpecialWindow 特有的动作之前，调用了当前对象的基类部分的一份拷贝的 Window::onResize。如果 Window::onResize 改变了当前对象（可能性并不小，因为 onResize 是一个 non-const 成员函数），当前对象并不会改变。作为替代，那个对象的一份拷贝被改变。如果 SpecialWindow::onResize 改变了当前对象，无论如何，当前对象将被改变，导致的境况是那些代码使当前对象进入一种病态，没有做基类的变更，却做了派生类的变更。

解决方法就是消除强制转型，用你真正想表达的来代替它。你不应该哄骗编译器将 *this 当作一个基类对象来处理，你应该调用当前对象的 onResize 的基类版本。就是这样：

```
class SpecialWindow: public Window {
public:
  virtual void onResize() {
    Window::onResize();                    // call Window::onResize
    ...                                    // on *this
  }
  ...

};
```

这个例子也表明如果你发现自己要做强制转型，这就是你可能做错了某事的一个信号。在你想用 dynamic_cast 时尤其如此。

在探究 dynamic_cast 的设计意图之前，值得留意的是很多 dynamic_cast 的实现都相当慢。例如，至少有一种通用的实现部分地基于对类名字进行字符串比较。如果你在一个位于四层深的单继承体系中的对象上执行 dynamic_cast，在这样一个实现下的每一个 dynamic_cast 都要付出相当于四次调用 strcmp 来比较类名字的成本。对于一个更深的或使用了多继承的继

承体系，付出的代价会更加昂贵。一些实现用这种方法工作是有原因的（它们不得不这样做以支持动态链接）。尽管如此，除了在普遍意义上警惕强制转型外，在性能敏感的代码中，你应该特别警惕 dynamic_casts。

对 dynamic_cast 的需要通常发生在这种情况下：你要在一个你确信为派生类的对象上执行派生类的操作，但是你只能通过一个基类的指针或引用来操控这个对象。有两个一般的方法可以避免这个问题。

第一个，使用存储着直接指向派生类对象的指针（通常是智能指针——参见 Item 13）的容器，从而消除通过基类接口操控这个对象的需要。例如，如果在我们的 Window/SpecialWindow 继承体系中，只有 SpecialWindows 支持 blinking，对于这样的做法：

```
class Window { ... };

class SpecialWindow: public Window {
public:
  void blink();
  ...
};
typedef                                          // see Item 13 for info
  std::vector<std::tr1::shared_ptr<Window> > VPW;  // on tr1::shared_ptr

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin();         // undesirable code:
     iter != winPtrs.end();                        // uses dynamic_cast
     ++iter) {
  if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))
     psw->blink();
}
```

设法用如下方法代替：

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;

VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin();        // better code: uses
     iter != winPtrs.end();                        // no dynamic_cast
     ++iter)
  (*iter)->blink();
```

当然，这个方法不允许你在同一个容器中存储所有可能的 Window 的派生类的指针。为了与不同的窗口类型一起工作，你可能需要多个类型安全（type-safe）的容器。

一个候选方法可以让你通过一个基类的接口操控所有可能的 Window 派生类，就是在基类中提供一个让你做你想做的事情的虚函数。例如，尽管只有 SpecialWindows 能 blink，在基类中声明这个函数，并提供一个什么都不做的缺省实现或许是有意义的：

```
class Window {
public:
  virtual void blink() {}                       // default impl is no-op;
  ...                                           // see Item 34 for why
};                                              // a default impl may be
                                                // a bad idea

class SpecialWindow: public Window {
public:
  virtual void blink() { ... }                  // in this class, blink
  ...                                           // does something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                    // container holds
                                                // (ptrs to) all possible
...                                             // Window types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                                    // note lack of
  (*iter)->blink();                             // dynamic_cast
```

无论哪种方法——使用类型安全的容器或在继承体系中上移虚函数——都不是到处适用的，但在很多情况下，它们提供了 dynamic_casting 之外另一个可行的候选方法。当它们可用时，你应该加以利用。

你应该绝对避免的一件东西就是包含了极联 dynamic_casts 的设计，也就是说，看起来类似这样的任何东西：

```
class Window { ... };

...                                     // derived classes are defined here

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end(); ++iter)
{
  if (SpecialWindow1 *psw1 =
       dynamic_cast<SpecialWindow1*>(iter->get())) { ... }

  else if (SpecialWindow2 *psw2 =
            dynamic_cast<SpecialWindow2*>(iter->get())) { ... }

  else if (SpecialWindow3 *psw3 =
            dynamic_cast<SpecialWindow3*>(iter->get())) { ... }

  ...
}
```

这样的 C++ 会生成的代码又大又慢，而且很脆弱，因为每次 Window 类继承体系发生变化，所有这样的代码都要必须被检查，以确认是否需要更新。（例如，如果增加了一个新的派生类，在上面的极联中或许就需要加入一个新的条件分支。）看起来类似这样的代码应该总是用基于虚函数的调用的某种东西来替换。

好的 C++ 极少使用强制转型，但在通常情况下完全去除也不实际。例如，第 118 页从 int 到 double 的强制转型，就是对强制转型的合理运用，虽然它并不是绝对必要。（那些代码应该被重写，声明一个新的类型为 double 的变量，并用 x 的值进行初始化。）就像大多数可疑的结构成分，强制转型应该被尽可能地隔离，典型情况是隐藏在函数内部，用函数的接口保护调用者远离内部的污秽的工作。

Things to Remember

- 避免强制转型的随时应用，特别是在性能敏感的代码中应用 dynamic_casts，如果一个设计需要强制转型，设法开发一个没有强制转型的侯选方案。
- 如果必须要强制转型，设法将它隐藏在一个函数中。客户可以用调用那个函数来代替在他们自己的代码中加入强制转型。
- 尽量用 C++ 风格的强制转型替换旧风格的强制转型。它们更容易被注意到，而且他们做的事情也更加明确。

Item 28: 避免返回对象内部构件的“句柄”

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设你正在一个包含矩形的应用程序上工作。每一个矩形都可以用它的左上角和右下角表示出来。为了将一个 Rectangle 对象保持在较小状态，你可能决定那些点的定义的域不应该包含在 Rectangle 本身之中，更合适的做法是放在一个由 Rectangle 指向的辅助的结构体中：

```
class Point {                      // class for representing points
public:
  Point(int x, int y);
  ...

  void setX(int newVal);
  void setY(int newVal);
  ...
};

struct RectData {                    // Point data for a Rectangle
  Point ulhc;                        // ulhc = " upper left-hand corner"
  Point lrhc;                        // lrhc = " lower right-hand corner"
};

class Rectangle {
  ...

private:
  std::tr1::shared_ptr<RectData> pData;          // see Item 13 for info on
};                                               // tr1::shared_ptr
```

由于 Rectangle 的客户需要有能力操控 Rectangle 的区域，因此类提供了 upperLeft 和 lowerRight 函数。可是，Point 是一个用户定义类型，所以，留心 Item 20 关于在典型情况下，以传引用的方式传递用户定义类型比传值的方式更加高效的观点，这些函数返回引向底层 Point 对象的引用：

```
class Rectangle {
public:
  ...
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

这个设计可以编译，但它是错误的。实际上，它是自相矛盾的。一方面，upperLeft 和 lowerRight 是被声明为 const 的成员函数，因为它们被设计成仅仅给客户提供一个获得 Rectangle 的点的方法，而不允许客户改变这个 Rectangle（参见 Item 3）。另一方面，两个函数都返回引向私有的内部数据的引用——调用者可以利用这些引用修改内部数据！例如：

```
Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);     // rec is a const rectangle from
                                         // (0, 0) to (100, 100)

rec.upperLeft().setX(50);                // now rec goes from
                                         // (50, 0) to (100, 100)!
```

请注意这里，upperLeft 的调用者是怎样利用返回的 rec 的内部 Point 数据成员的引用来改变这个成员的。但是 rec 却被期望为 const！

这直接引出两条经验。第一，一个数据成员被封装，但是具有最高可访问级别的函数还是能够返回引向它的引用。在当前情况下，虽然 ulhc 和 lrhc 被声明为 private，它们还是被有效地公开了，因为 public 函数 upperLeft 和 lowerRight 返回了引向它们的引用。第二，如果一个 const 成员函数返回一个引用，引向一个与某个对象有关并存储在这个对象本身之外的数据，这个函数的调用者就可以改变那个数据（这正是二进制位常量性的局限性（参见 Item 3）的一个副作用）。

我们前面做的每件事都涉及到成员函数返回的引用，但是，如果它们返回指针或者迭代器，因为同样的原因也会存在同样的问题。引用，指针，和迭代器都是句柄（handle）（持有其它对象的方法），而返回一个对象内部构件的句柄总是面临危及对象封装安全的风险。就像我们看到的，它同时还能导致 const 成员函数改变了一个对象的状态。

我们通常认为一个对象的“内部构件”就是它的数据成员，但是不能被常规地公开访问的成员函数（也就是说，它是 protected 或 private 的）也是对象内部构件的一部分。同样地，不要返回它们的句柄也很重要。这就意味着你绝不应该有一个成员函数返回一个指向拥有较小的可访问级别的成员函数的指针。如果你这样做了，它的可访问级别就会与那个拥有较大的可访问级别的函数相同，因为客户能够得到指向这个拥有较小的可访问级别的函数的指针，然后就可以通过这个指针调用这个函数。

无论如何，返回指向成员函数的指针的函数是难得一见的，所以让我们把注意力返回到 Rectangle 类和它的 upperLeft 和 lowerRight 成员函数。我们在这些函数中挑出来的问题都只需简单地将 const 用于它们的返回类型就可以排除：

```
class Rectangle {
public:
  ...
  const Point& upperLeft() const { return pData->ulhc; }
  const Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

通过这个修改的设计，客户可以读取定义一个矩形的 Points，但他们不能写它们。这就意味着将 upperLeft 和 upperRight 声明为 const 不再是一句空话，因为他们不再允许调用者改变对象的状态。至于封装的问题，我们总是故意让客户看到做成一个 Rectangle 的 Points，所

以这是封装的一个故意的放松之处。更重要的，它是一个有限的放松：只有读访问是被这些函数允许的，写访问依然被禁止。

虽然如此，upperLeft 和 lowerRight 仍然返回一个对象内部构件的句柄，而这有可能造成其它方面的问题。特别是，这会导致空悬句柄：引用了不再存在的对象的构件的句柄。这种消失的对象的最普通的来源就是函数返回值。例如，考虑一个函数，返回在一个矩形窗体中的 GUI 对象的 bounding box：

```
class GUIObject { ... };

const Rectangle                              // returns a rectangle by
  boundingBox(const GUIObject& obj);         // value; see Item 3 for why
                                             // return type is const
```

现在，考虑客户可能会这样使用这个函数：

```
GUIObject *pgo;                              // make pgo point to
...                                          // some GUIObject

const Point *pUpperLeft =                    // get a ptr to the upper
  &(boundingBox(*pgo).upperLeft());          // left point of its
                                             // bounding box
```

对 boundingBox 的调用会返回一个新建的临时的 Rectangle 对象。这个对象没有名字，所以我们就称它为 temp。于是 upperLeft 就在 temp 上被调用，这个调用返回一个引向 temp 的一个内部构件的引用，特别是，它是由 Points 构成的。随后 pUpperLeft 指向这个 Point 对象。到此为止，一切正常，但是我们无法继续了，因为在这个语句的末尾，boundingBox 的返回值—— temp ——被销毁了，这将间接导致 temp 的 Points 的析构。接下来，剩下 pUpperLeft 指向一个已经不再存在的对象；pUpperLeft 空悬在创建它的语句的末尾！

这就是为什么任何返回一个对象的内部构件的句柄的函数都是危险的。它与那个句柄是指针，引用，还是迭代器没什么关系。它与是否受到 cosnt 的限制没什么关系。它与那个成员函数返回的句柄本身是否是 const 没什么关系。全部的问题在于一个句柄被返回了，因为一旦这样做了，你就面临着这个句柄比它引用的对象更长寿的风险。

这并不意味着你永远不应该让一个成员函数返回一个句柄。有时你必须如此。例如，operator[] 允许你从 string 和 vector 中取出单独的元素，而这些 operator[]s 就是通过返回引向容器中的数据的引用来工作的（参见 Itme 3）——当容器本身被销毁，数据也将销毁。尽管如此，这样的函数属于特例，而不是惯例。

Things to Remember

- 避免返回对象内部构件的句柄（引用，指针，或迭代器）。这样会提高封装性，帮助 const 成员函数产生 cosnt 效果，并将空悬句柄产生的可能性降到最低

Item 29: 争取异常安全（exception-safe）的代码

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

异常安全（Exception safety）有点像怀孕（pregnancy）……但是，请把这个想法先控制一会儿。我们还不能真正地议论生育（reproduction），直到我们排除万难渡过求爱时期（courtship）。（此段作者使用的 3 个词均有双关含义，pregnancy 也可理解为富有意义，reproduction 也可理解为再现，再生，courtship 也可理解为争取，谋求。为了与后面的译文对应，故按照现在的译法。——译者注）

假设我们有一个类，代表带有背景图像的 GUI 菜单。这个类被设计成在多线程环境中使用，所以它有一个用于并行控制（concurrency control）的互斥体（mutex）：

```
class PrettyMenu {
public:
  ...
  void changeBackground(std::istream& imgSrc);           // change background
  ...                                                    // image

private:

  Mutex mutex;                       // mutex for this object

  Image *bgImage;                    // current background image
  int imageChanges;                  // # of times image has been changed
};
```

考虑这个 PrettyMenu 的 changeBackground 函数的可能的实现：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  lock(&mutex);                         // acquire mutex (as in Item 14)

  delete bgImage;                       // get rid of old background
  ++imageChanges;                       // update image change count
  bgImage = new Image(imgSrc);          // install new background

  unlock(&mutex);                       // release mutex
}
```

从异常安全的观点看，这个函数烂到了极点。异常安全有两条要求，而这里全都没有满足。

当一个异常被抛出，异常安全的函数应该：

- 没有资源泄露。上面的代码没有通过这个测试，因为如果 "new Image(imgSrc)" 表达式产生一个异常，对 unlock 的调用就永远不会执行，而那个互斥体也将被永远挂起。

- 不允许数据结构恶化。如果 "new Image(imgSrc)" 抛出异常，bgImage 被遗留下来指向一个被删除对象。另外，尽管并没有将一张新的图像设置到位，imageChanges 也已经被增加。（在另一方面，旧的图像被明确地删除，所以我料想你会争辩说图像已经被“改变”了。）

规避资源泄露问题比较容易，因为 Item 13 解释了如何使用对象管理资源，而 Item 14 又引进了 Lock 类作为一种时尚的确保互斥体被释放的方法：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);                 // from Item 14: acquire mutex and
                                   // ensure its later release
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

关于像 Lock 这样的资源管理类的最好的事情之一是它们通常会使函数变短。看到对 unlock 的调用不再需要了吗？作为一个一般的规则，更少的代码就是更好的代码。因为在改变的时候这样可以较少误入歧途并较少产生误解。

随着资源泄露被我们甩在身后，我们可以把我们的注意力集中到数据结构恶化。在这里我们有一个选择，但是在我们能选择之前，我们必须先面对定义我们的选择的术语。

异常安全函数提供下述三种保证之一：

- 函数提供基本保证（the basic guarantee），允诺如果一个异常被抛出，程序中剩下的每一件东西都处于合法状态。没有对象或数据结构被破坏，而且所有的对象都处于内部调和状态（所有的类不变量都被满足）。然而，程序的精确状态可能是不可预期的。例如，我们可以重写 changeBackground，以致于如果一个异常被抛出，PrettyMenu 对象可以继续保留原来的背景图像，或者它可以持有某些缺省的背景图像，但是客户无法预知到底是哪一个。（为了查明这一点，他们大概必须调用某个可以告诉他们当前背景图像是什么的成员函数。）

- 函数提供强力保证（the strong guarantee），允诺如果一个异常被抛出，程序的状态不会发生变化。调用这样的函数在感觉上是极其微弱的，如果它们成功了，它们就完全成功，如果它们失败了，程序的状态就像它们从没有被调用过一样。

  与提供强力保证的函数一起工作比与只提供基本保证的函数一起工作更加容易，因为调用提供强力保证的函数之后，仅有两种可能的程序状态：像预期一样成功执行了函数，或者继续保持函数被调用时当时的状态。与之相比，如果调用只提供基本保证的函数引发了异常，程序可能存在于任何合法的状态。

- 函数提供不抛出保证（the nothrow guarantee），允诺决不抛出异常，因为它们只做它们答应要做的。所有对内建类型（例如，ints，指针，等等）的操作都是不抛出（nothrow）的（也就是说，提供不抛出保证）。这是异常安全代码中必不可少的基础构件。

> 假定一个带有空的异常规格（exception specification）的函数是不抛出的似乎是合理的，但这不一定正确的。例如，考虑这个函数：
>
> ```
> int doSomething() throw(); // note empty exception spec.
> ```
>
> 这并不是说 doSomething 永远不会抛出异常；而是说如果 doSomething 抛出一个异常，它就是一个严重的错误，应该调用 unexpected 函数 [1]。实际上，doSomething 可能根本不提供任何异常保证。一个函数的声明（如果有的话，也包括它的异常规格（exception specification））不能告诉你一个函数是否正确，是否可移植，或是否高效，而且，即便有，它也不能告诉你它会提供哪一种异常安全保证。所有这些特性都由函数的实现决定，而不是它的声明能决定的。
>
> [1] 关于 unexpected 函数的资料，可以求助于你中意的搜索引擎或包罗万象的 C++ 课本。（你或许有幸搜到 set_unexpected，这个函数用于指定 unexpected 函数。）

异常安全函数必须提供上述三种保证中的一种。如果它没有提供，它就不是异常安全的。于是，选择就在于决定你写的每一个函数究竟要提供哪种保证。除非要处理遗留下来的非异常安全的代码（本 Item 稍后我们要讨论这个问题），只有当你的最高明的需求分析团队为你的应用程序识别出的一项需求就是泄漏资源以及运行于被破坏的数据结构之上时，不提供异常安全保证才能成为一个选项。

作为一个一般性的规则，你应该提供实际可达到的最强力的保证。从异常安全的观点看，不抛出的函数（nothrow functions）是极好的，但是在 C++ 的 C 部分之外部不调用可能抛出异常的函数简直就是寸步难行。使用动态分配内存的任何东西（例如，所有的 STL 容器）如果不能找到足够的内存来满足一个请求（参见 Item 49），在典型情况下，它就会抛出一个 bad_alloc 异常。只要你能做到就提供不抛出保证，但是对于大多数函数，选择是在基本的保证和强力的保证之间的。

在 changeBackground 的情况下，提供差不多的强力保证并不困难。首先，我们将 PrettyMenu 的 bgImage 数据成员的类型从一个内建的 Image* 指针改变为 Item 13 中描述的智能资源管理指针中的一种。坦白地讲，在预防资源泄漏的基本原则上，这完全是一个好主意。它帮助我们提供强大的异常安全保证的事实进一步加强了 Item 13 的论点——使用对象（诸如智能指针）管理资源是良好设计的基础。在下面的代码中，我展示了 tr1::shared_ptr 的使用，因为当进行通常的拷贝时它的更符合直觉的行为使得它比 auto_ptr 更可取。

第二，我们重新排列 changeBackground 中的语句，以致于直到图像发生变化，才增加 imageChanges。这是一个很好的策略——直到某件事情真正发生了，再改变一个对象的状态来表示某事已经发生。

这就是修改之后的代码：

```
class PrettyMenu {
  ...
  std::tr1::shared_ptr<Image> bgImage;
  ...
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  Lock ml(&mutex);

  bgImage.reset(new Image(imgSrc));  // replace bgImage's internal
                                     // pointer with the result of the
                                     // "new Image" expression
  ++imageChanges;
}
```

注意这里不再需要手动删除旧的图像，因为在智能指针内部已经被处理了。此外，只有当新的图像被成功创建了删除行为才会发生。更准确地说，只有当 tr1::shared_ptr::reset 函数的参数（"new Image(imgSrc)" 的结果）被成功创建了，这个函数才会被调用。只有在对 reset 的调用的内部才会使用 delete，所以如果这个函数从来不曾进入，delete 就从来不曾使用。同样请注意一个管理资源（动态分配的 Image）的对象（tr1::shared_ptr）的使用再次缩短了 changeBackground 的长度。

正如我所说的，这两处改动差不多有能力使 changeBackground 提供强力异常安全保证。美中不足的是什么呢？参数 imgSrc。如果 Image 的构造函数抛出一个异常，输入流（input stream）的读标记（read marker）可能已经被移动，而这样的移动就成为对程序的其它部分来说可见的一个状态的变化。直到 changeBackground 着手解决这个问题之前，它只能提供基本异常安全保证。

无论如何，让我们把它放在一边，并且依然假装 changeBackground 可以提供强力保证。（我相信你至少能用一种方法做到这一点，或许可以通过将它的参数从一个 istream 改变到包含图像数据的文件的文件名。）有一种通常的设计策略可以有代表性地产生强力保证，而且熟悉它是非常必要的。这个策略被称为 "copy and swap"。它的原理很简单。先做出一个你要改变的对象的拷贝，然后在这个拷贝上做出全部所需的改变。如果改变过程中的某些操作抛出了异常，最初的对象保持不变。在所有的改变完全成功之后，将被改变的对象和最初的对象在一个不会抛出异常的操作中进行交换。

这通常通过下面的方法实现：将每一个对象中的全部数据从“真正的”对象中放入到一个单独的实现对象中，然后将一个指向实现对象的指针交给真正对象。这通常被称为 "pimpl idiom"，Item 31 描述了它的一些细节。对于 PrettyMenu 来说，它一般就像这样：

```
struct PMImpl {                               // PMImpl = "PrettyMenu
  std::tr1::shared_ptr<Image> bgImage;        // Impl."; see below for
  int imageChanges;                           // why it's a struct
};

class PrettyMenu {
  ...

private:
  Mutex mutex;
  std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
  using std::swap;                            // see Item 25

  Lock ml(&mutex);                            // acquire the mutex

  std::tr1::shared_ptr<PMImpl>                // copy obj. data
    pNew(new PMImpl(*pImpl));

  pNew->bgImage.reset(new Image(imgSrc));     // modify the copy
  ++pNew->imageChanges;

  swap(pImpl, pNew);                          // swap the new
                                              // data into place

}                                             // release the mutex
```

在这个例子中，我选择将 PMImpl 做成一个结构体，而不是类，因为通过让 pImpl 是 private 就可以确保 PrettyMenu 数据的封装。将 PMImpl 做成一个类虽然有些不那么方便，却没有增加什么好处。（这也会使有面向对象洁癖者走投无路。）如果你愿意，PMImpl 可以嵌套在 PrettyMenu 内部，像这样的打包问题与我们这里所关心的写异常安全的代码的问题没有什么关系。

copy-and-swap 策略是一种全面改变或丝毫不变一个对象的状态的极好的方法，但是，在通常情况下，它不能保证全部函数都是强力异常安全的。为了弄清原因，考虑一个 changeBackground 的抽象化身—— someFunc，它使用了 copy-and-swap，但是它包含了对另外两个函数（f1 和 f2）的调用：

```
void someFunc()
{
  ...                                     // make copy of local state
  f1();
  f2();
  ...                                     // swap modified state into place
}
```

很明显，如果 f1 或 f2 低于强力异常安全，someFunc 就很难成为强力异常安全的。例如，假设 f1 仅提供基本保证。为了让 someFunc 提供强力保证，它必须写代码在调用 f1 之前测定整个程序的状态，并捕捉来自 f1 的所有异常，然后恢复到最初的状态。

即使 f1 和 f2 都是强力异常安全的，事情也好不到哪去。如果 f1 运行完成，程序的状态已经发生了毫无疑问的变化，所以如果随后 f2 抛出一个异常，即使 f2 没有改变任何东西，程序的状态也已经和调用 someFunc 时不同。

问题在于副作用。只要函数仅对局部状态起作用（例如，someFunc 仅仅影响调用它的那个对象的状态），它提供强力保证就相对容易。当函数的副作用影响了非局部数据，它就会困难得多。例如，如果调用 f1 的副作用是改变数据库，让 someFunc 成为强力异常安全就非常困难。一般情况下，没有办法撤销已经提交的数据库变化，其他数据库客户可能已经看见了数据库的新状态。

类似这样的问题会阻止你为函数提供强力保证，即使你希望去做。另一个问题是效率。copy-and-swap 的要点是这样一个想法：改变一个对象的数据的拷贝，然后在一个不会抛出异常的操作中将被改变的数据和原始数据进行交换。这就需要做出每一个要改变的对象的拷贝，这可能会用到你不能或不情愿动用的时间和空间。强力保证是非常值得的，当它可用时你应该提供它，除非在它不能 100% 可用的时候。

当它不可用时，你就必须提供基本保证。在实践中，你可能会发现你能为某些函数提供强力保证，但是效率和复杂度的成本使得它难以支持大量的其它函数。无论何时，只要你作出过一个提供强力保证的合理的成果，就没有人会因为你仅仅提供了基本保证而站在批评你的立场上。对于很多函数来说，基本保证是一个完全合理的选择。

如果你写了一个根本没有提供异常安全保证的函数，事情就不同了，因为在这一点上有罪推定是合情合理的，直到你证明自己是清白的。你应该写出异常安全的代码。除非你能做出有说服力的答辩。请再次考虑 someFunc 的实现，它调用了函数 f1 和 f2。假设 f2 根本没有提供异常安全保证，甚至没有基本保证。这就意味着如果 f2 发生一个异常，程序可能会在 f2 内部泄漏资源。这也意味着 f2 可能会恶化数据结构，例如，已排序数组可能不再排序，一个正在从一个数据结构传送到另一个数据结构去的对象可能丢失，等等。没有任何办法可以让 someFunc 能弥补这些问题。如果 someFunc 调用的函数不提供异常安全保证，someFunc 本身就不能提供任何保证。

请允许我回到怀孕。一个女性或者怀孕或者没有。局部怀孕是绝不可能的。与此相似，一个软件或者是异常安全的或者不是。没有像一个局部异常安全的系统这样的东西。一个系统即使只有一个函数不是异常安全的，那么系统作为一个整体就不是异常安全的，因为调用那个函数可能发生泄漏资源和恶化数据结构。不幸的是，很多 C++ 的遗留代码在写的时候没有留意异常安全，所以现在的很多系统都不是异常安全的。它们混合了用非异常安全（exception-unsafe）的方式书写的代码。

没有理由让事情的这种状态永远持续下去。当书写新的代码或改变现存代码时，要仔细考虑如何使它异常安全。以使用对象管理资源开始。（还是参见 Item 13。）这样可以防止资源泄漏。接下来，决定三种异常安全保证中的哪一种是你实际上能够为你写的每一个函数提供的最强的保证，只有当你不调用遗留代码就别无选择的时候，才能满足于没有保证。既是为你的函数的客户也是为了将来的维护人员，文档化你的决定。一个函数的异常安全保证是它的接口的可见部分，所以你应该特意选择它，就像你特意选择一个函数接口的其它方面。

四十年前，到处都是 goto 的代码被尊为最佳实践。现在我们为书写结构化控制流程而奋斗。二十年前，全局可访问数据被尊为最佳实践。现在我们为封装数据而奋斗，十年以前，写函数时不必考虑异常的影响被尊为最佳实践。现在我们为写异常安全的代码而奋斗。

时光在流逝。我们生活着。我们学习着。

Things to Remember

- 即使当异常被抛出时，异常安全的函数不会泄露资源，也不允许数据结构被恶化。这样的函数提供基本的，强力的，或者不抛出保证。
- 强力保证经常可以通过 copy-and-swap 被实现，但是强力保证并非对所有函数都可用。
- 一个函数通常能提供的保证不会强于他所调用的函数中最弱的保证。

Item 30: 理解 inline 化的介入和排除

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

inline 函数——多么棒的主意啊！它们看起来像函数，它们产生的效果也像函数，它们在各方面都比宏好得太多太多（参见 Item 2），而你却可以在调用它们时不招致函数调用的成本。你还有什么更多的要求呢？

实际上你得到的可能比你想的更多，因为避免函数调用的成本只是故事的一部分。在典型情况下，编译器的优化是为了一段连续的没有函数调用的代码设计的，所以当你 inline 化一个函数，你可能就使得编译器能够对函数体实行上下文相关的特殊优化。大多数编译器都不会对 "outlined" 函数调用实行这样的优化。

然而，在编程中，就像在生活中，没有免费午餐，而 inline 函数也不例外。一个 inline 函数背后的思想是用函数本体代替每一处对这个函数的调用，而且不必拿着统计表中的 Ph.D. 就可以看出这样可能会增加你的目标代码的大小。在有限内存的机器上，过分热衷于 inline 化会使得程序对于可用空间来说过于庞大。即使使用了虚拟内存，inline 引起的代码膨胀也会导致附加的分页调度，减少指令缓存命中率，以及随之而来的性能损失。

在另一方面，如果一个 inline 函数本体很短，为函数本体生成的代码可能比为一个函数调用生成的代码还要小。如果是这种情况，inline 化这个函数可以实际上导致更小的目标代码和更高的指令缓存命中率！

记住，inline 是向编译器发出的一个请求，而不是一个命令。这个请求能够以显式的或隐式的方式提出。隐式的方法就是在一个类定义的内部定义一个函数：

```
class Person {
public:
  ...
  int age() const { return theAge; }    // an implicit inline request: age is
  ...                                   // defined in a class definition

private:
  int theAge;
};
```

这样的函数通常是成员函数，但是 Item 46 解释了友元函数也能被定义在类的内部，如果它们在那里，它们也被隐式地声明为 inline。

显式地声明一个 inline 函数的方法是在它的声明之前加上 inline 关键字。例如，以下就是标准 max 模板（来自 <algorithm>）经常用到的的实现方法：

```
template<typename T>                               // an explicit inline
inline const T& std::max(const T& a, const T& b)   // request: std::max is
{ return a < b ? b : a; }                          // preceded by "inline"
```

max 是一个模板的事实引出一个观察结论：inline 函数和模板一般都是定义在头文件中的。这就使得一些程序员得出结论断定函数模板必须是 inline。这个结论是非法的而且有潜在的危害，所以它值得我们考察一下。

inline 函数一般必须在头文件内，因为大多数构建环境在编译期间进行 inline 化。为了用被调用函数的函数本体替换一个函数调用，编译器必须知道函数看起来像什么样子。（有一些构建环境可以在连接期间进行 inline 化，还有少数几个——比如，基于 .NET Common Language Infrastructure (CLI) 的控制环境——居然能在运行时 inline 化。然而，这些环境都是例外，并非规则。inline 化在大多数 C++ 程序中是一个编译时行为。）

模板一般在头文件内，因为编译器需要知道一个模板看起来像什么以便用到它时对它进行实例化。（同样，也不是全部如此。一些构建环境可以在连接期间进行模板实例化。然而，编译期实例化更为普遍。）

模板实例化与 inline 化无关。如果你写了一个模板，而且你认为所有从这个模板实例化出来的函数都应该是 inline 的，那么就声明这个模板为 inline，这就是上面的 std::max 的实现被做的事情。但是如果你为没有理由要 inline 化的函数写了一个模板，就要避免声明这个模板为 inline（无论显式的还是隐式的）。inline 化是有成本的，而且你不希望在毫无预见的情况下遭遇它们。我们已经说到 inline 化是如何引起代码膨胀的（这对于模板作者来说是极为重要的一个考虑事项——参见 Item 44），但是，还有其它的成本，过一会儿我们再讨论。

在做这件事之前，我们先来完成对这个结论的考察：inline 是一个编译器可能忽略的请求。大多数编译器拒绝它们认为太复杂的 inline 函数（例如，那些包含循环或者递归的），而且，除了最细碎的以外的全部虚拟函数的调用都不会被 inline 化。不应该对这后一个结论感到惊讶。虚拟意味着“等待，直到运行时才能断定哪一个函数被调用”，而 inline 意味着“执行之前，用被调用函数取代调用的地方”。如果编译器不知道哪一个函数将被调用，你很难责备它们拒绝 inline 化这个函数本体。

所有这些加在一起，得出：一个被指定的 inline 函数是否能真的被 inline 化，取决于你所使用的构建环境——主要是编译器。幸运的是，大多数编译器都有一个诊断层次，在它们不能 inline 化一个你提出的函数时，会导致一个警告（参见 Item 53）。

有时候，即使当编译器完全心甘情愿地 inline 化一个函数，他们还是会为这个 inline 函数生成函数本体。例如，如果你的程序要持有一个 inline 函数的地址，编译器必须为它生成一个 outlined 函数本体。他们怎么能生成一个指向根本不存在的函数的指针呢？再加上，编译器一般不会对通过函数指针的调用进行 inline 化，这就意味着，对一个 inline 函数的调用可能被也可能不被 inline 化，依赖于这个调用是如何做成的：

```
inline void f() {...}      // assume compilers are willing to inline calls to f

void (*pf)() = f;          // pf points to f

...

f();                       // this call will be inlined, because it's a "normal" call
pf();                      // this call probably won't be, because it's through
                           // a function pointer
```

甚至在你从来没有使用函数指针的时候，未 inline 化的 inline 函数的幽灵也会时不时地拜访你，因为程序员并不必然是函数指针的唯一需求者。有时候编译器会生成构造函数和析构函数的 out-of-line 拷贝，以便它们能得到指向这些函数的指针，在对数组中的对象进行构造和析构时使用。

事实上，构造函数和析构函数对于 inline 化来说经常是一个比你在不经意的检查中所能显示出来的更加糟糕的候选者。例如，考虑下面这个类 Derived 的构造函数：

```
class Base {
public:
 ...

private:
   std::string bm1, bm2;               // base members 1 and 2
};

class Derived: public Base {
public:
  Derived() {}                         // Derived's ctor is empty — or is it?
  ...

private:
  std::string dm1, dm2, dm3;           // derived members 1–3
};
```

这个构造函数看上去像一个 inline 化的极好的候选者，因为它不包含代码。但是视觉会被欺骗。

C++ 为对象被创建和被销毁时所发生的事情做出了各种保证。例如，当你使用 new 时，你的动态的被创建对象会被它们的构造函数自动初始化，而当你使用 delete。则相应的析构函数会被调用。当你创建一个对象时，这个对象的每一个基类和每一个数据成员都会自动构造，而当一个对象被销毁时，则发生关于析构的反向过程。如果在一个对象构造期间有一个异常被抛出，这个对象已经完成构造的任何部分都被自动销毁。所有这些情节，C++ 只说什么必须发生，但没有说如何发生。那是编译器的实现者的事，但显然这些事情不会自己发生。在你的程序中必须有一些代码使这些事发生，而这些代码——由编译器写出的代码和在编译期间插入你的程序的代码——必须位于某处。有时它们最终就位于构造函数和析构函数中，所以我们可以设想实现为上面那个声称为空的 Derived 的构造函数生成的代码就相当于下面这样：

```
Derived::Derived()                         // conceptual implementation of
{                                          // "empty" Derived ctor

 Base::Base();                             // initialize Base part

 try { dm1.std::string::string(); }        // try to construct dm1
 catch (...) {                             // if it throws,
   Base::~Base();                          // destroy base class part and
   throw;                                  // propagate the exception
 }

 try { dm2.std::string::string(); }        // try to construct dm2
 catch(...) {                              // if it throws,
   dm1.std::string::~string();             // destroy dm1,
   Base::~Base();                          // destroy base class part, and
   throw;                                  // propagate the exception
 }

 try { dm3.std::string::string(); }        // construct dm3
 catch(...) {                              // if it throws,
   dm2.std::string::~string();             // destroy dm2,
   dm1.std::string::~string();             // destroy dm1,
   Base::~Base();                          // destroy base class part, and
   throw;                                  // propagate the exception
 }
}
```

这些代码并不代表真正的编译器所生成的，因为真正的编译器会用更复杂的方法处理异常。尽管如此，它还是准确地反映了 Derived 的“空”构造函数必须提供的行为。不论一个编译器的异常多么复杂，Derived 的构造函数至少必须调用它的数据成员和基类的构造函数，而这些调用（它们自己也可能是 inline 的）会影响它对于 inline 化的吸引力。

同样的原因也适用于 Base 的构造函数，所以如果它是 inline 的，插入它的全部代码也要插入 Derived 的构造函数（通过 Derived 的构造函数对 Base 的构造函数的调用）。而且如果 string 的构造函数碰巧也是 inline 的，Derived 的构造函数中将增加五个那个函数代码的拷贝，分别对应于 Derived 对象中的五个 strings（两个继承的加上三个它自己声明的）。也许在现在，为什么说是否 inline 化 Derived 的构造函数不是一个不经大脑的决定就很清楚了。类似的考虑也适用于 Derived 的析构函数，用同样的或者不同的方法，必须保证所有被 Derived 的构造函数初始化的对象被完全销毁。

库设计者必须评估声明函数为 inline 的影响，因为为库中的客户可见的 inline 函数提供二进制升级版本是不可能的。换句话说，如果 f 是一个库中的一个 inline 函数，库的客户将函数 f 的本体编译到他们的应用程序中。如果一个库的实现者后来决定修改 f，所有使用了 f 的客户都必须重新编译。这常常会令人厌烦。在另一方面，如果 f 是一个非 inline 函数，对 f 的改变只需要客户重新连接。这与重新编译相比显然减轻了很大的负担，而且，如果库中包含的函数是动态链接的，这就是一种对于用户来说完全透明的方法。

为了程序开发的目标，在头脑中牢记这些需要考虑的事项是很重要的，但是从编码期间的实用观点来看，占有支配地位的事实是：大多数调试器会与 inline 函数发生冲突。这不应该是什么重大的发现。你怎么能在一个不在那里的函数中设置断点呢？虽然一些构建环境设法支持 inline 函数的调试，多数环境还是简单地为调试构建取消了 inline 化。

这就导出了一个用于决定哪些函数应该被声明为 inline，哪些不应该的合乎逻辑的策略。最初，不要 inline 任何东西，或者至少要将你的 inline 化的范围限制在那些必须 inline 的（参见 Item 46）和那些实在微不足道的（就像第 135 页上的 Person::age）函数上。通过慎重地使用 inline，你可以使调试器的使用变得容易，但是你也将 inline 化放在了它本来应该在的地位：作为一种手动的优化。不要忘记由经验确定的 80-20 规则，它宣称一个典型的程序用 80% 的时间执行 20% 的代码。这是一个重要的规则，因为它提醒你作为一个软件开发者的目标是识别出能全面提升你的程序性能的 20% 的代码。你可以 inline 或者用其他方式无限期地调节你的函数，但除非你将精力集中在正确的函数上，否则就是白白浪费精力。

Things to Remember

- 将大部分 inline 限制在小的，调用频繁的函数上。这使得程序调试和二进制升级更加容易，最小化潜在的代码膨胀，并最大化提高程序速度的几率。
- 不要仅仅因为函数模板出现在头文件中，就将它声明为 inline。

Item 31: 最小化文件之间的编译依赖

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

你进入到你的程序中，并对一个类的实现进行了细微的改变。提醒你一下，不是类的接口，只是实现，仅仅是 private 的东西。然后你重建（rebuild）这个程序，预计这个任务应该只花费几秒钟。毕竟只有一个类被改变。你在 Build 上点击或者键入 make（或者其它等价行为），接着你被惊呆了，继而被郁闷，就像你突然意识到整个世界都被重新编译和连接！当这样的事情发生的时候，你不讨厌它吗？

问题在于 C++ 没有做好从实现中剥离接口的工作。一个类定义不仅指定了一个类的接口而且有相当数量的实现细节。例如：

```
class Person {
public:
  Person(const std::string& name, const Date& birthday,
         const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;
  ...

private:
      std::string theName;        // implementation detail
      Date theBirthDate;          // implementation detail
      Address theAddress;         // implementation detail
};
```

在这里，如果不访问 Person 的实现使用到的类，也就是 string，Date 和 Address 的定义，类 Person 就无法编译。这样的定义一般通过 #include 指令提供，所以在定义 Person 类的文件中，你很可能会找到类似这样的东西：

```
#include <string>
#include "date.h"
#include "address.h"
```

不幸的是，这样就建立了定义 Person 的文件和这些头文件之间的编译依赖关系。如果这些头文件中的一些发生了变化，或者这些头文件所依赖的文件发生了变化，包含 Person 类的文件和使用了 Person 的文件一样必须重新编译，这样的层叠编译依赖关系为项目带来数不清的麻烦。

你也许想知道 C++ 为什么坚持要将一个类的实现细节放在类定义中。例如，你为什么不能这样定义 Person，单独指定这个类的实现细节呢？

```
namespace std {
     class string;              // forward declaration (an incorrect
}                               // one — see below)

class Date;                     // forward declaration
class Address;                  // forward declaration

class Person {
public:
      Person(const std::string& name, const Date& birthday,
                 const Address& addr);
      std::string name() const;
      std::string birthDate() const;
      std::string address() const;
    ...
};
```

如果这样可行，只有在类的接口发生变化时，Person 的客户才必须重新编译。

这个主意有两个问题。第一个，string 不是一个类，它是一个 typedef (for basic_string<char>)。造成的结果就是，string 的前向声明（forward declaration）是不正确的。正确的前向声明要复杂得多，因为它包括另外的模板。然而，这还不是要紧的，因为你不应该试着手动声明标准库的部件。作为替代，直接使用适当的 #includes 并让它去做。标准头文件不太可能成为编译的瓶颈，特别是在你的构建环境允许你利用预编译头文件时。如果解析标准头文件真的成为一个问题。你也许需要改变你的接口设计，避免使用导致不受欢迎的 #includes 的标准库部件。

第二个（而且更重要的）难点是前向声明的每一件东西必须让编译器在编译期间知道它的对象的大小。考虑：

```
int main()
{
 int x;                // define an int

 Person p( params );   // define a Person
   ...
}
```

当编译器看到 x 的定义，它们知道它们必须为保存一个 int 分配足够的空间（一般是在栈上）。这没什么问题，每一个编译器都知道一个 int 有多大。当编译器看到 p 的定义，它们知道它们必须为一个 Person 分配足够的空间，但是它们怎么推测出一个 Person 对象有多大呢？它们得到这个信息的唯一方法是参考这个类的定义，但是如果一个省略了实现细节的类定义是合法的，编译器怎么知道要分配多大的空间呢？

这个问题在诸如 Smalltalk 和 Java 这样的语言中就不会发生，因为，在这些语言中，当一个类被定义，编译器仅仅为一个指向一个对象的指针分配足够的空间。也就是说，它们处理上面的代码就像这些代码是这样写的：

```
int main()
{
  int x;                // define an int

  Person *p;            // define a pointer to a Person
  ...
}
```

当然，这是合法的 C++，所以你也可以自己来玩这种“将类的实现隐藏在一个指针后面”的游戏。对 Person 做这件事的一种方法就是将它分开到两个类中，一个仅仅提供一个接口，另一个实现这个接口。如果那个实现类名为 PersonImpl，Person 就可以如此定义：

```
#include <string>                      // standard library components
                                       // shouldn't be forward-declared

#include <memory>                      // for tr1::shared_ptr; see below

class PersonImpl;                      // forward decl of Person impl. class
class Date;                            // forward decls of classes used in

class Address;                         // Person interface
class Person {
public:
 Person(const std::string& name, const Date& birthday,
        const Address& addr);
 std::string name() const;
 std::string birthDate() const;
 std::string address() const;
 ...

private:                                    // ptr to implementation;
  std::tr1::shared_ptr<PersonImpl> pImpl;  // see Item 13 for info on
};                                          // std::tr1::shared_ptr
```

这样，主类（Person）除了一个指向它的实现类（PersonImpl）的指针（这里是一个 tr1::shared_ptr ——参见 Item 13）之外不包含其它数据成员。这样一个设计经常被说成是使用了 pimpl 惯用法（指向实现的指针 "pointer to implementation"）。在这样的类中，那个指针的名字经常是 pImpl，就像上面那个。

用这样的设计，使 Person 的客户脱离 dates，addresses 和 persons 的细节。这些类的实现可以随心所欲地改变，但 Person 的客户却不必重新编译。另外，因为他们看不到 Person 的实现细节，客户就不太可能写出以某种方式依赖那些细节的代码。这就是接口和实现的真正分离。

这个分离的关键就是用对声明的依赖替代对定义的依赖。这就是最小化编译依赖的精髓：只要能实现，就让你的头文件独立自足，如果不能，就依赖其它文件中的声明，而不是定义。其它每一件事都从这个简单的设计策略产生。所以：

- 当对象的引用和指针可以做到时就避免使用对象。仅需一个类型的声明，你就可以定义到这个类型的引用或指针。而定义一个类型的对象必须要存在这个类型的定义。
- 只要你能做到，就用对类声明的依赖替代对类定义的依赖。注意你声明一个使用一个类的函数时绝对不需要有这个类的定义，即使这个函数通过传值方式传递或返回这个类：

```
class Date;                           // class declaration

Date today();                         // fine — no definition
void clearAppointments(Date d);       // of Date is needed
```

当然，传值通常不是一个好主意（参见 Item 20），但是如果你发现你自己因为某种原因而使用它，依然不能为引入不必要的编译依赖辩解。

不声明 Date 就可以声明 today 和 clearAppointments 的能力可能会令你感到惊奇，但是它其实并不像看上去那么不同寻常。如果有人调用这些函数，则 Date 的定义必须在调用之前被看到。为什么费心去声明没有人调用的函数，你想知道吗？很简单。并不是没有人调用它们，而是并非每个人都要调用它们。如果你有一个包含很多函数声明的库，每一个客户都要调用每一个函数是不太可能的。通过将提供类定义的责任从你的声明函数的头文件转移到客户的包含函数调用的文件，你就消除了客户对他们并不真的需要的类型的依赖。

- 为声明和定义分别提供头文件。为了便于坚持上面的指导方针，头文件需要成对出现：一个用于声明，另一个用于定义。当然，这些文件必须保持一致。如果一个声明在一个地方被改变了，它必须在两处都被改变。得出的结果是：库的客户应该总是 #include 一个声明文件，而不是自己前向声明某些东西，而库的作者应该提供两个头文件。例如，想要声明 today 和 clearAppointments 的 Date 的客户不应该像前面展示的那样手动前向声明 Date。更合适的是，它应该 #include 适当的用于声明的头文件：

```
#include "datefwd.h"            // header file declaring (but not
                               // defining) class Date

Date today();                   // as before
void clearAppointments(Date d);
```

仅有声明的头文件的名字 "datefwd.h" 基于来自标准 C++ 库（参见 Item 54）的头文件 <iosfwd>。<iosfwd> 包含 iostream 组件的声明，而它们相应的定义在几个不同的头文件中，包括 <sstream>，<streambuf>，<fstream> 和 <iostream>。

<iosfwd> 在其它方面也有启发意义，而且它解释了本 Item 的建议对于模板和非模板一样有效。尽管 Item 30 解释了在很多构建环境中，模板定义的典型特征是位于头文件中，但有些环境允许模板定义在非头文件中，所以为模板提供一个仅有声明的头文件依然是有意义的。<iosfwd> 就是一个这样的头文件。

C++ 还提供了 export 关键字允许将模板声明从模板定义中分离出来。不幸的是，支持 export 的编译器非常少，而与 export 打交道的实际经验就更少了。结果是，现在就说 export 在高效 C++ 编程中扮演什么角色还为时尚早。

像 Person 这样的使用 pimpl 惯用法的类经常被称为 Handle 类。为了避免你对这样的类实际上做什么事的好奇心，一种方法是将所有对他们的函数调用都转送给相应的实现类，而使用实现类来做真正的工作。例如，这就是两个 Person 的成员函数可以被如何实现的例子：

```
#include "Person.h"          // we're implementing the Person class,
                             // so we must #include its class definition

#include "PersonImpl.h"      // we must also #include PersonImpl's class
                             // definition, otherwise we couldn't call
                             // its member functions; note that
                             // PersonImpl has exactly the same
                             // member functions as Person — their
                             // interfaces are identical

Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}

std::string Person::name() const
{
  return pImpl->name();
}
```

注意 Person 的成员函数是如何调用 PersonImpl 的成员函数的（通过使用 new ——参见 Item 16），以及 Person::name 是如何调用 PersonImpl::name 的。这很重要。使 Person 成为一个 Handle 类不需要改变 Person 要做的事情，仅仅是改变了它做事的方法。

另一个不同于 Handle 类的候选方法是使 Person 成为一个被叫做 Interface 类的特殊种类的抽象基类。这样一个类的作用是为派生类指定一个接口（参见 Item 34）。结果，它的典型特征是没有数据成员，没有构造函数，有一个虚析构函数（参见 Item 7）和一组指定接口的纯虚函数。

Interface 类类似 Java 和 .NET 中的 Interfaces，但是 C++ 并不会为 Interface 类强加那些 Java 和 .NET 为 Interfaces 强加的种种约束。例如，Java 和 .NET 都不允许 Interfaces 中有数据成员和函数实现，但是 C++ 不禁止这些事情。C++ 的巨大弹性是有用处的。就像 Item 36 解释的，在一个继承体系的所有类中非虚拟函数的实现应该相同，因此将这样的函数实现为声明它们的 Interface 类的一部分就是有意义的。

一个 Person 的 Interface 类可能就像这样：

```
class Person {
public:
  virtual ~Person();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
  virtual std::string address() const = 0;
  ...
};
```

这个类的客户必须针对 Person 的指针或引用编程，因为实例化包含纯虚函数的类是不可能的。（然而，实例化从 Person 派生的类是可能的——参见后面。）和 Handle 类的客户一样，除非 Interface 类的接口发生变化，否则 Interface 类的客户不需要重新编译。

一个 Interface 类的客户必须有办法创建新的对象。他们一般通过调用一个为“可以真正实例化的派生类”扮演构造函数的角色的函数做到这一点的。这样的函数一般称为 factory 函数（参见 Item 13）或虚拟构造函数（virtual constructors）。他们返回指向动态分配的支持 Interface 类的接口的对象的指针（智能指针更合适——参见 Item 18）。这样的函数在 Interface 类内部一般声明为 static：

```
class Person {
public:
 ...

 static std::tr1::shared_ptr<Person>    // return a tr1::shared_ptr to a new
   create(const std::string& name,      // Person initialized with the
          const Date& birthday,         // given params; see Item 18 for
          const Address& addr);         // why a tr1::shared_ptr is returned
 ...
};
```

客户就像这样使用它们：

```
std::string name;
Date dateOfBirth;
Address address;
...

// create an object supporting the Person interface
std::tr1::shared_ptr<Person> pp(Person::create(name, dateOfBirth, address));

...

std::cout << pp->name()                 // use the object via the
          << " was born on "            // Person interface
          << pp->birthDate()
          << " and now lives at "
          << pp->address();
...                                     // the object is automatically
                                        // deleted when pp goes out of
                                        // scope — see Item 13
```

当然，在某些地点，必须定义支持 Interface 类的接口的具体类并调用真正的构造函数。这所有的一切发生的场合，在那个文件中所包含虚拟构造函数的实现之后的地方。例如，Interface 类 Person 可以有一个提供了它继承到的虚函数的实现的具体的派生类 RealPerson：

```
class RealPerson: public Person {
public:
  RealPerson(const std::string& name, const Date& birthday,
             const Address& addr)
  : theName(name), theBirthDate(birthday), theAddress(addr)
  {}

  virtual ~RealPerson() {}

  std::string name() const;        // implementations of these
  std::string birthDate() const;   // functions are not shown, but
  std::string address() const;     // they are easy to imagine

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};
```

对这个特定的 RealPerson，写 Person::create 确实没什么价值：

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            const Date& birthday,
                                            const Address& addr)
{
  return std::tr1::shared_ptr<Person>(new RealPerson(name, birthday,addr));
}
```

Person::create 的一个更现实的实现会创建不同派生类型的对象，依赖于诸如，其他函数的参数值，从文件或数据库读出的数据，环境变量等等。

RealPerson 示范了两个最通用的实现一个 Interface 类机制之一：从 Interface 类（Person）继承它的接口规格，然后实现接口中的函数。实现一个 Interface 类的第二个方法包含多继承（multiple inheritance），在 Item 40 中探讨这个话题。

Handle 类和 Interface 类从实现中分离出接口，因此减少了文件之间的编译依赖。如果你是一个喜好挖苦的人，我知道你正在找小号字体写成的限制。“所有这些把戏会骗走我什么呢？”你小声嘀咕着。答案是计算机科学中非常平常的：它会消耗一些运行时的速度，再加上每个对象的一些额外的内存。

在 Handle 类的情况下，成员函数必须通过实现的指针得到对象的数据。这就在每次访问中增加了一个间接层。而且你必须在存储每一个对象所需的内存量中增加这一实现的指针的大小。最后，这一实现的指针必须被初始化（在 Handle 类的构造函数中）为指向一个动态分配的实现的对象，所以你要承受动态内存分配（以及随后的释放）所固有的成本和遭遇 bad_alloc (out-of-memory) 异常的可能性。

对于 Interface 类，每一个函数调用都是虚拟的，所以你每调用一次函数就要支付一个间接跳转的成本（参见 Item 7）。还有，从 Interface 派生的对象必须包含一个 virtual table 指针（还是参见 Item 7）。这个指针可能增加存储一个对象所需的内存的量，依赖于这个 Interface 类是否是这个对象的虚函数的唯一来源。

最后，无论 Handle 类还是 Interface 类都不能在 inline 函数的外面大量使用。Item 30 解释了为什么函数本体一般必须在头文件中才能做到 inline，但是 Handle 类和 Interface 类一般都设计成隐藏类似函数本体这样的实现细节。

然而，因为它们所涉及到的成本而简单地放弃 Handle 类和 Interface 类会成为一个严重的错误。虚拟函数也是一样，但你还是不能放弃它们，你能吗？（如果能，你看错书了。）作为替代，考虑以一种改进的方式使用这些技术。在开发过程中，使用 Handle 类和 Interface 类来最小化实现发生变化时对客户的影响。当能看出在速度和/或大小上的不同足以证明增加类之间的耦合是值得的时候，可以用具体类取代 Handle 类和 Interface 类供产品使用。

Things to Remember

- 最小化编译依赖后面的一般想法是用对声明的依赖取代对定义的依赖。基于此想法的两个方法是 Handle 类和 Interface 类。
- 库头文件应该以完整并且只有声明的形式存在。无论是否包含模板都适用于这一点。

Item 32: 确保 public inheritance 模拟 "is-a"

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在 Some Must Watch While Some Must Sleep (W. H. Freeman and Company, 1974) 这本书中，William Dement 讲述了一个他试图让他的学生的记住他的课程中最重要的东西的故事。书中声称，他告诉他的班级，一般的英国中小学生对于 1066 年发生的 Hastings 战争的历史并没有什么了解。他着重强调，如果一个孩子记住了一点儿什么的，他或者她也就是记住了 1066 这个年代。对于上他的课程的学生，Dement 滔滔不绝地讲，其中只有很少的重要信息，包括安眠药却引起了失眠这样充满趣味的事情。他希望他的学生即使忘记课程中讨论的其它每一件事，也能记住这些很少的重大事件，而且他在整个学期再三地回顾这些基础的内容。

在课程结束的时候，期末考试的最后一道题是："写下从这个课程中得到的，你一生都将确切地记住的一件事。"当 Dement 给这次考试打分的时候，他几乎晕了过去。几乎每一个人都写了"1066"。

因此，我一再煞费苦心地向你宣扬，使用 C++ 语言进行 object-oriented programming 时唯一最重要规则就是：public inheritance（公开继承）意味着 "is-a"。要让这个规则刻骨铭心。

如果你写了一个 class D ("Derived") 从 class B ("Base") 公开继承，你就是在告诉 C++ 编译器（以及你的代码的读者）每一个类型为 D 的对象也是一个类型为 B 的对象，但是反之则不然。你就是在说 B 描绘了一个比 D 更一般的概念，D 描述了一个比 B 更特殊的概念。你就是在声称一个类型为 B 的对象可以使用的任何地方，一个类型为 D 的对象一样可以使用，因为每一个类型为 D 的对象也就是一个类型为 B 的对象。另一方面，如果你需要一个类型为 D 的对象，一个类型为 B 的对象则不行：每一个 D 都是一个 B，但是反之则不然。

C++ 坚持对 public inheritance 的这一解释。考虑这个例子：

```
class Person {...};

class Student: public Person {...};
```

我们从日常的经验知道每一个学生都是一个人，但并不是每一个人都是一个学生。这就是由这个继承体系严格确定的意义。我们期望每一件对于人来说成立的事情——例如，他或她有一个出生日——对于一个学生来说也成立。我们不期望每一件对于学生来说成立的事情——例如，他或她在一所特定的学校注册——对于普通人来说也成立。一个人的概念比一个学生的概念更普通，一个学生一个专门类型的人。

在 C++ 领域中，任何期望引数类型为 Person（或 pointer-to-Person 或 reference-to-Person）的函数都可以接受一个 Student object（或 pointer-to-Student 或 reference-to-Student）：

```
void eat(const Person& p);            // anyone can eat

void study(const Student& s);         // only students study

Person p;                             // p is a Person
Student s;                            // s is a Student

eat(p);                               // fine, p is a Person

eat(s);                               // fine, s is a Student,
                                      // and a Student is-a Person

study(s);                             // fine

study(p);                             // error! p isn't a Student
```

这一点只对 public inheritance 才成立。只有 Student 以 public 方式从 Person 派生，C++ 才有我所描述的行为。private inheritance 意味着完全不同的其它事情（参见 Item 39），而 protected inheritance 究竟意味什么使我困惑至今。

public inheritance 和 is-a 等价听起来简单，但有时你的直觉会误导你。例如，企鹅是一种鸟没有问题，而鸟能飞也没有问题。如果我们天真地试图用 C++ 来表达，我们就会得到：

```
class Bird {
public:
  virtual void fly();                  // birds can fly

  ...
};

class Penguin:public Bird {            // penguins are birds
  ...
};
```

突然间我们遇到了麻烦，因为这个继承体系表示企鹅能飞，我们知道这不是真的。发生了什么呢？

在这种情况下，我们成了不严谨的语言——英语的牺牲品。当我们说鸟能飞的时候，我们的意思并非是说所有种类的鸟都能飞，我们不过是说，大体上，鸟有飞的能力。如果我们说得更准确些，我们应该承认有几种不能飞的鸟，并提出如下继承体系，它对事实的模拟要好得多：

```
class Bird {
  ...                                       // no fly function is declared
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class Penguin: public Bird {

  ...                                       // no fly function is declared

};
```

这个继承体系比最初的设计更忠实于我们真正知道的东西。

至此我们还是没有完全做好关于这些鸟的事情，因为对于某些软件系统来说，可能并不需要区分能飞的和不能飞的鸟。如果你的应用程序对于鸟喙和鸟翼做了很多处理，而不打算对飞行做什么处理的话，最初的 two-class 的继承体系可能完全适用。它是对“没有一个适用于所有软件的完美设计”这样的事实的一个简单反映。最好的设计依赖于系统究竟期望做什么，无论现在还是未来。如果你的程序对飞行一无所知，而且也不期望以后能知道些什么，那么不分辨能飞与不能飞的鸟可能就是一个非常完美的设计决策。事实上，它可能比区分它们的设计更为可取，因为你试图模拟的世界中就没有这样一种区分。。

对于如何处理我所说的“所有的鸟能飞，企鹅是鸟，企鹅不能飞，啊……哦……”的问题，还有另一种思想观念。那就是为企鹅重定义 fly 函数，以便让它产生一个运行时错误。

```
void error(const std::string& msg);       // defined elsewhere

class Penguin: public Bird {
public:
  virtual void fly() { error("Attempt to make a penguin fly!");}

  ...

};
```

认可“这里所说的一些事情与你所想的可能不同”是很重要的。这不是说“企鹅不能飞”。而是说“企鹅能飞，但对它试图真的这样做就是一个错误”。

你能说出其中的区别吗？从错误被发觉时间方面看。“企鹅不能飞”的禁令可以由编译器强令执行，但是对“让企鹅真的去飞是一个错误”的规约的违反，只有在运行时才能被发觉。

为了表达“企鹅不能飞”这个限制，你要确保不要为 Penguin 对象定义这样的函数：

```
class Bird {
  ...                                 // no fly function is declared
};
class Penguin: public Bird {
  ...                                 // no fly function is declared
};
```

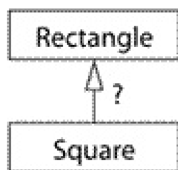
如果你现在试图让企鹅飞，编译器将因为你的违例而惩罚你：

```
Penguin p;
p.fly();                      // error!
```

这与你采用产生运行时错误的方法，得到完全不同的行为。使用那个方法，关于对 p.fly 的调用，编译器一言不发。Item 18 解释了好的接口可以在编译时防止非法代码，所以你应该用通过编译器阻止企鹅飞翔企图的设计代替只在运行时检测的设计

也许你会承认你的鸟类学知识可能不足，但是你对自己对基本几何学的掌握很自信，是吗？我的意思是说，矩形和正方形能有多么复杂？

好吧，回答这个简单的问题：应该让 class Square 从 class Rectangle 公开继承吗？



“咄！”你说，“当然应该！每一个人都知道一个正方形就是一个矩形，但是反过来就不一定了。”这完全正确，至少在学校里是。但是我不认为我们现在还在学校里。

考虑如下代码：

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;               // return current values
  virtual int width() const;

  ...

};

void makeBigger(Rectangle& r)               // function to increase r's area
{
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);               // add 10 to r's width

  assert(r.height() == oldHeight);          // assert that r's
}                                           // height is unchanged
```

很清楚，断言应该永远不会失败。makeBigger 仅仅改变了 r 的宽度，它的高度始终没有变化。

现在，考虑以下代码，使用 public inheritance 使得 squares 可以像 rectangles 一样进行处理：

```
class Square: public Rectangle {...};

Square s;

...

assert(s.width() == s.height());           // this must be true for all squares

makeBigger(s);                             // by inheritance, s is-a Rectangle,
                                           // so we can increase its area

assert(s.width() == s.height());           // this must still be true
                                           // for all squares
```

和刚才那个一样明显，第二个断言也应该永远不会失败。根据定义，正方形的宽度和高度是相等的。

但是，现在有一个问题，我们怎样才能协调以下断言？

- 调用 makeBigger 之前，s 的高度和它的宽度相等；
- 在 makeBigger 内，s 的宽度发生变化，但是它的高度没有变化；
- 从 makeBigger 返回之后，s 的高度还要和它的宽度相等。（注意 s 是通过 by reference 方式传入 makeBigger 的，所以 makeBigger 能改变 s 自身，而不是 s 的拷贝。）

嘟嘟？

欢迎来到 public inheritance 的奇妙世界，你在其它学习领域——包括数学——中发展起来的本能，可能不再像你所期望的那样帮助你。在这种情况下，基本的难点在于一些适用于矩形（它的宽度可以独立于他的高度而自行变化）的事情不适用于正方形（它的宽度和高度必须相等）。但是 public inheritance 断言，适用于 base class objects（基类对象）的每一件事——每一件事！——也适用于 derived class objects（派生类对象）。在矩形和正方形的情况下（还有 Item 38 中一个包含 sets 和 lists 的例子），这个断言失效，所以用 public inheritance 模拟它们的关系是完全错误的。编译器允许你这样做，但是就像我们已经看到的，它不能保证代码的行为正确。每一个程序员都必须认识到，仅仅通过编译的代码，并不意味着它可以工作。

不必忧虑你在过去这些年发展起来的软件直觉在你走近 object-oriented design 时失效。那些知识依然是有价值的，但是现在你应该在你的设计候选武器库中加入 inheritance，你还必须在你的直觉中加入新的洞察力来指导你正确使用 inheritance。当某个人向你展示一个几页长的函数时，你可能会及时地从 Penguin 继承自 Bird 或 Square 继承自 Rectangle 得到有趣的感觉。它可能是接近事实的正确方法，只是不太像而已。

is-a 关系并不是能存在于两个 classes 之间的唯一关系。另外两个常见的 inter-class 关系是 "has-a" 和 "is-implemented-in-terms-of"。这些关系将在 Item 38 和 39 中考虑。因为用这些其它重要关系中的一个来不正确地模拟 is-a 而造成的 C++ 设计错误并不罕见，所以你应该确保你理解了这些关系之间的不同，并知道在 C++ 中如何才能用它们做最好的模拟。

Things to Remember

- public inheritance 意味着 "is-a"。适用于 base classes 的每一件事也适用于 derived classes，因为每一个 derived class object 都是一个 base class object。

Item 33: 避免覆盖（hiding）“通过继承得到的名字”

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

莎士比亚有一个关于名字的说法。"What's in a name?" 他问道，"A rose by any other name would smell as sweet."（语出《罗密欧与朱丽叶》第二幕第二场，朱生豪先生译为："姓名本来是没有意义的；我们叫做玫瑰的这一种花，要是换了个名字，他的香味还是同样的芬芳。"梁实秋先生译为："姓算什么？我们所谓有玫瑰，换个名字，还是一样的香。"——译者注）。莎翁也写过 "he that filches from me my good name ... makes me poor indeed."（语出《奥塞罗》第三幕第三场，朱生豪先生译为："可是谁偷去了我的名誉，那么他虽然并不因此而富足，我却因为失去它而成为赤贫了。"梁实秋先生译为："但是他若夺去我的名誉，于他不见有利，对我却是一件损失哩。"——译者注）。好吧，在 C++ 中，我们该用哪种态度对待通过继承得到的名字呢？

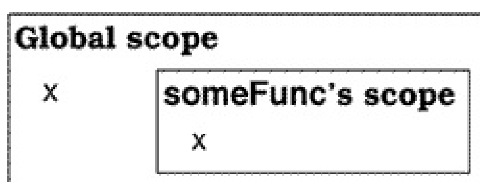
事情的实质与继承没什么关系。它与作用域有关。我们都知道它在代码中是这样的，

```
int x;                        // global variable

void someFunc()
{
  double x;                   // local variable

  std::cin >> x;              // read a new value for local x
}
```

读入 x 的语句指涉 local 变量 x，而不是 global 变量 x，因为内层作用域的名字覆盖（"遮蔽"）外层作用域的名字。我们可以像这样形象地表示作用域的状况：



当编译器在 someFunc 的作用域中遇到名字 x 时，他们巡视 local 作用域看看是否有什么东西叫这个名字。因为那里有，它们就不再检查其它作用域。在此例中，someFunc 的 x 类型为 double，而 global x 类型为 int，但这不要紧。C++ 的 name-hiding 规则仅仅是覆盖那个名字。而相对应的名字的类型是否相同是无关紧要的。在此例中，一个名为 x 的 double 覆盖了一个名为 x 的 int。

加入 inheritance 以后。我们知道当我们在一个 derived class member function 内指涉位于 base class 内的一件东西（例如，一个 member function，一个 typedef，或者一个 data member）时，编译器能够找到我们指涉的东西是因为 derived classes 继承到声明于 base classes 中的东西。实际中的运作方法是将 derived class 的作用域嵌套在 base class 作用域之中。例如：

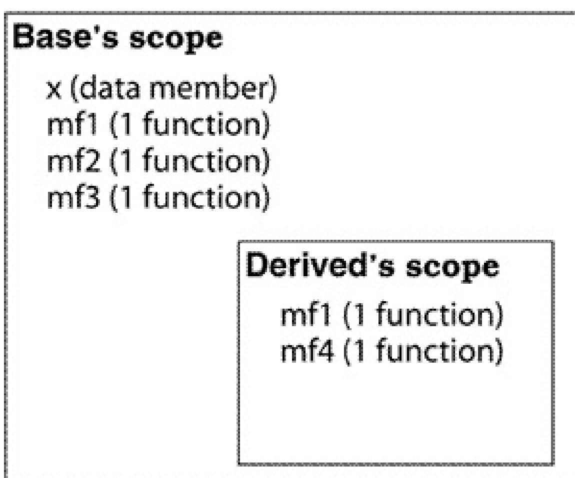
```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf2();
  void mf3();

  ...
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf4();

  ...
};
```



本例中包含的既有 public 名字也有 private 名字，既有 data members 也有 member functions。member functions 既有 pure virtual 的，也有 simple (impure) virtual 的，还有 non-virtual 的。那是为了强调我们谈论的事情是关于名字的。例子中还可以包括其它类型的名字，例如，enums，nested classes，和 typedefs。在这里的讨论中唯一重要的事情是“它们是名字”。与它们是什么东西的名字毫不相关。这个示例中使用了 single inheritance，但是一旦你理解了在 single inheritance 下会发生什么，C++ 在 multiple inheritance 下的行为就很容易预见了。

假设 mf4 在 derived class 中被实现，其中一部分，如下：

```
void Derived::mf4()
{

  ...
  mf2();

  ...
}
```

当编译器看到这里对名字 mf2 的使用，它就必须断定它指涉什么。它通过搜索名为 mf2 的某物的定义的作用域来做这件事。首先它在 local 作用域中搜索（也就是 mf4 的作用域），但是它没有找到被称为 mf2 的任何东西的声明。然后它搜索它的包含作用域，也就是 class Derived 的作用域。它依然没有找到叫做 mf2 的任何东西，所以它上移到它的上一层包含作用域，也就是 base class 的作用域。在那里它找到了名为 mf2 的东西，所以搜索停止。如果在 Base 中没有 mf2，搜索还会继续，首先是包含 Base 的 namespace(s)（如果有的话），最后是 global 作用域。

我刚刚描述的过程虽然是正确的，但它还不是一个关于 C++ 中名字如何被找到的完整的描述。无论如何，我们的目的不是为了充分了解关于写一个编译器时的名字搜索问题。而是为了充分了解如何避免令人吃惊的意外，而对于这个任务，我们已经有了大量的信息。

再次考虑前面的示例，而且这一次我们 overload mf1 和 mf3，并且为 Derived 增加一个 mf3 的版本。（就像 Item 36 解释的，Derived 对 mf3 ——一个通过继承得到的 non-virtual function ——的重载，使得这个设计立即变得可疑，但是出于对 inheritance 之下名字可见性问题的关心，我们就装作没看见。）

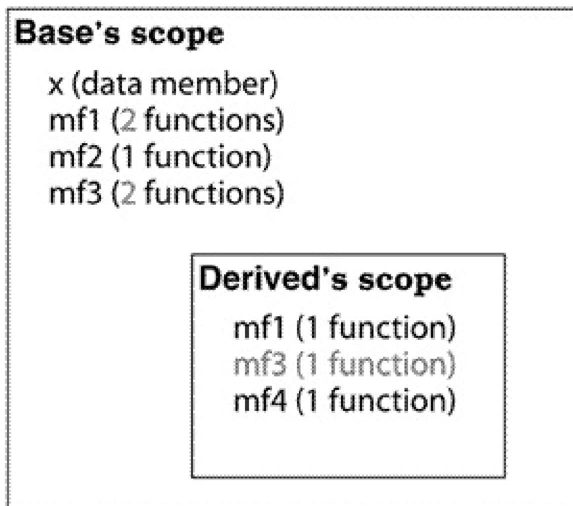
```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```



以上代码导致的行为会使每一个第一次遇到它的 C++ 程序员吃惊。基于作用域的名字覆盖规则（scope-based name hiding rule）不会有什么变化，所以 base class 中的所有名为 mf1 和 mf3 的函数被 derived class 中的名为 mf1 和 mf3 的函数覆盖。从名字搜索的观点看，Base::mf1 和 Base::mf3 不再被 Derived 继承！

```
Derived d;
int x;

...
d.mf1();                    // fine, calls Derived::mf1
d.mf1(x);                   // error! Derived::mf1 hides Base::mf1
d.mf2();                    // fine, calls Base::mf2

d.mf3();                    // fine, calls Derived::mf3
d.mf3(x);                   // error! Derived::mf3 hides Base::mf3
```

就像你看到的，即使 base 和 derived classes 中的函数具有不同的参数类型，它也同样适用，而且不管函数是 virtual 还是 non-virtual，它也同样适用。与“在本 Item 的开始处，函数 someFunc 中的 double x 覆盖了 global 作用域中的 int x”的道理相同，这里 Derived 中的函数 mf3 覆盖了具有不同类型的名为 mf3 的一个 Base 函数。

这一行为背后的根本原因是为了防止“当你在一个 library 或者 application framework 中创建一个新的 derived class 时，偶然地发生从遥远的 base classes 继承 overloads 的情况”。不幸的是，一般情况下你是需要继承这些 overloads 的。实际上，如果你使用了 public inheritance 而又没有继承这些 overloads，你就违反了 Item 32 讲解的“base 和 derived classes 之间是 is-a 关系”这一 public inheritance 的基本原则。在这种情况下，你几乎总是要绕过 C++ 对“通过继承得到的名字”的缺省的覆盖机制。

你可以用 using declarations 做到这一点：

```
class Base {
private:
  int x;

public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  virtual void mf2();

  void mf3();
  void mf3(double);
  ...
};

class Derived: public Base {
public:
  using Base::mf1;       // make all things in Base named mf1 and mf3
  using Base::mf3;       // visible (and public) in Derived's scope

  virtual void mf1();
  void mf3();
  void mf4();
  ...
};
```

Base's scope

x (data member)
mf1 (2 functions)
mf2 (1 function)
mf3 (2 functions)

Derived's scope

mf1 (2 functions)
mf3 (2 functions)
mf4 (1 function)

现在 inheritance 就可以起到预期的作用：

```
Derived d;
int x;

...

d.mf1();                 // still fine, still calls Derived::mf1
d.mf1(x);                // now okay, calls Base::mf1

d.mf2();                 // still fine, still calls Base::mf2

d.mf3();                 // fine, calls Derived::mf3
d.mf3(x);                // now okay, calls Base::mf3
```

这意味着如果你从一个带有重载函数的 base class 继承，而且你只想重定义或替换它们中的一部分，你需要为每一个你不想覆盖的名字使用 using declaration。如果你不这样做，一些你希望继承下来的名字会被覆盖。

可以想象在某些时候你不希望从你的 base classes 继承所有的函数。在 public inheritance 中，这是绝不会发生的，这还是因为，它违反了 public inheritance 在 base 和 derived classes 之间的 is-a 关系。（这就是为什么上面的 using declarations 在 derived class 的 public 部分：在 base class 中是 public 的名字在公有继承的 derived class 中也应该是 public。）然而，在 private inheritance（参见 Item 39）中，它还是有意义的。例如，假设 Derived 从 Base 私有继承，而且 Derived 只想继承没有参数的那个 mf1 的版本。在这里，using declaration 没有这个本事，因为一个 using declaration 会使得所有具有给定名字的函数在 derived class 中可见。不，这里是使用了一种不同的技术的情形，即，一个简单的 forwarding function（转调函数）：

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  ...                                    // as before
};

class Derived: private Base {
public:
  virtual void mf1()                   // forwarding function; implicitly
  { Base::mf1(); }                     // inline (see Item 30)
  ...
};

...

Derived d;
int x;

d.mf1();                               // fine, calls Derived::mf1
d.mf1(x);                              // error! Base::mf1() is hidden
```

forwarding function（转调函数）的另一个功效是用于老式的编译器，它们（不正确地）不支持用 using declarations 将“通过继承得到的名字”引入到 derived class 的作用域。

这就是关于 inheritance 和 name hiding 的全部故事，但是当 inheritance 与 templates 结合起来，“通过继承得到的名字被隐藏”的问题会以一种全然不同的形式呈现出来。关于全部 angle-bracket-demarcated（边边角角）的细节，参见 Item 43。

Things to Remember

- derived classes 中的名字覆盖 base classes 中的名字，在 public inheritance 中，这从来不是想要的。
- 为了使隐藏的名字重新可见，使用 using declarations 或者 forwarding functions（转调函数）。

Item 34: 区分 inheritance of interface（接口继承）和 inheritance of implementation（实现继承）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

(public) inheritance 这个表面上简单易懂的观念，一旦被近距离审视，就会被证明是由两个相互独立的部分组成的：inheritance of function interfaces（函数接口的继承）和 inheritance of function implementations（函数实现的继承）。这两种 inheritance 之间的差异正好符合本书 Introduction 中论述的 function declarations（函数声明）和 function definitions（函数定义）之间的差异。

作为一个 class 的设计者，有的时候你想要 derived classes 只继承一个 member function 的 interface (declaration)。有的时候你想要 derived classes 既继承 interface（接口）也继承 implementation（实现），但你要允许它们替换他们继承到的 implementation。还有的时候你想要 derived classes 继承一个函数的 interface（接口）和 implementation（实现），而不允许它们替换任何东西。

为了更好地感觉这些选择之间的不同之处，考虑一个在图形应用程序中表示几何图形的 class hierarchy（类继承体系）：

```
class Shape {
public:
  virtual void draw() const = 0;

  virtual void error(const std::string& msg);

  int objectID() const;

  ...
};

class Rectangle: public Shape { ... };

class Ellipse: public Shape { ... };
```

Shape 是一个 abstract class（抽象类），它的 pure virtual function（纯虚拟函数）表明了这一点。作为结果，客户不能创建 Shape class 的实例，只能创建从它继承的 classes 的实例。但是，Shape 对所有从它（公有）继承的类施加了非常强大的影响，因为

成员函数 interfaces are always inherited。就像 Item 32 解释的，public inheritance 意味着 is-a，所以对一个 base class 来说成立的任何东西，对于它的 derived classes 也必须成立。因此，如果一个函数适用于一个 class，它也一定适用于它的 derived classes。

Shape class 中声明了三个函数。第一个，draw，在一个明确的显示设备上画出当前对象。第二个，error，如果 member functions 需要报告一个错误，就调用它。第三个，objectID，返回当前对象的唯一整型标识符。每一个函数都用不同的方式声明：draw 是一个 pure virtual function（纯虚拟函数）；error 是一个 simple (impure?) virtual function（简单虚拟函数）；而 objectID 是一个 non-virtual function（非虚拟函数）。这些不同的声明暗示了什么呢？

考虑第一个 pure virtual function（纯虚拟函数）draw：

```
class Shape {
public:
  virtual void draw() const = 0;
  ...
};
```

pure virtual functions（纯虚拟函数）的两个最显著的特性是它们必须被任何继承它们的具体类重新声明，和抽象类中一般没有它们的定义。把这两个特性加在一起，你应该认识到

声明一个 pure virtual function（纯虚拟函数）的目的是使 derived classes 继承一个函数 interface only。

这就使 Shape::draw function 具有了完整的意义，因为它要求所有的 Shape 对象必须能够画出来是合情合理的，但是 Shape class 本身不能为这个函数提供一个合乎情理的缺省的实现。例如，画一个椭圆的算法和画一个矩形的算法是非常不同的，Shape::draw 的声明告诉具体 derived classes 的设计者："你必须提供一个 draw function，但是我对于你如何实现它不发表意见。"

顺便提一句，为一个 pure virtual function（纯虚拟函数）提供一个定义是有可能的。也就是说，你可以为 Shape::draw 提供一个实现，而 C++ 也不会抱怨什么，但是调用它的唯一方法是用 class name 限定修饰这个调用：

```
Shape *ps = new Shape;              // error! Shape is abstract

Shape *ps1 = new Rectangle;         // fine
ps1->draw();                     // calls Rectangle::draw

Shape *ps2 = new Ellipse;           // fine
ps2->draw();                     // calls Ellipse::draw

ps1->Shape::draw();                 // calls Shape::draw

ps2->Shape::draw();                 // calls Shape::draw
```

除了帮助你在鸡尾酒会上给同行程序员留下印象外，这个特性通常没什么用处，然而，就像下面你将看到的，它能用来作为一个"为 simple (impure) virtual functions 提供一个 safer-than-usual 的实现"的机制。

simple virtual functions 背后的故事和 pure virtuals 有一点不同。derived classes 照常还是继承函数的 interface，但是 simple virtual functions 提供了一个可以被 derived classes 替换的实现。如果你为此考虑一阵儿，你就会认识到

声明一个 simple virtual function 的目的是让 derived classes 继承一个函数 interface as well as a default implementation。

考虑 Shape::error 的情况：

```
class Shape {
public:
  virtual void error(const std::string& msg);
  ...
};
```

interface 要求每一个 class 必须支持一个在遭遇到错误时被调用的函数，但是每一个 class 可以自由地用它觉得合适的任何方法处理错误。如果一个 class 不需要做什么特别的事情，它可以仅仅求助于 Shape class 中提供的错误处理的缺省版本。也就是说，Shape::error 的声明告诉 derived classes 的设计者："你应该支持一个 error function，但如果你不想自己写，你可以求助 Shape class 中的缺省版本。"

结果是：允许 simple virtual functions 既指定一个函数接口又指定一个缺省实现是危险的。来看一下为什么，考虑一个 XYZ 航空公司的飞机的 hierarchy（继承体系）。XYZ 只有两种飞机，Model A 和 Model B，它们都严格地按照同样的方法飞行。于是，XYZ 设计如下 hierarchy（继承体系）：

```
class Airport { ... };                     // represents airports

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void Airplane::fly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
```

为了表述所有的飞机必须支持一个 fly 函数，并为了"不同机型可能（在理论上）需要不同的对 fly 的实现"的事实，Airplane::fly 被声明为 virtual。然而，为了避免在 ModelA 和 ModelB classes 中些重复的代码，缺省的飞行行为由 Airplane::fly 的函数体提供，供 ModelA 和 ModelB 继承。

这是一个经典的 object-oriented 设计。因为两个 classes 共享一个通用特性（它们实现 fly 的方法），所以这个通用特性就被转移到一个 base class 之中，并由两个 classes 来继承这个特性。这个设计使得通用特性变得清楚明白，避免了代码重复，提升了未来的可扩展性，简化了长期的维护——因为 object-oriented 技术，所有这些东西都受到很高的追捧。XYZ 航空公司应该引以为荣。

现在，假设 XYZ 公司的财富增长了，决定引进一种新机型，Model C。Model C 在某些方面与 Model A 和 Model B 不同。特别是，它的飞行不同。

XYZ 公司的程序员在 hierarchy（继承体系）中增加了 Model C 的 class，但是由于他们匆匆忙忙地让新的机型投入服务，他们忘记了重定义 fly function：

```
class ModelC: public Airplane {

  ...                                   // no fly function is declared
};
```

于是，在他们的代码中，就出现了类似这样的东西：

```
Airport PDX(...);                       // PDX is the airport near my home

Airplane *pa = new ModelC;

...

pa->fly(PDX);                          // calls Airplane::fly!
```

这是一个灾难：企图让一个 ModelC object 像一个 ModelA 或 ModelB 一样飞行。这在旅行人群中可不是一种鼓舞人心的行为。

这里的问题并不在于 Airplane::fly 有缺省的行为，而是在于 ModelC 被允许不必明确说出它要做什么就可以继承这一行为。幸运的是，“为 derived classes（派生类）提供缺省的行为，但是除非它们提出明确的要求，否则就不交给他们”是很容易做到的。这个诀窍就是切断 virtual function（虚拟函数）的 interface（接口）和它的 default implementation（缺省实现）之间的联系。以下用的就是这个方法：

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

protected:
  void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
  default code for flying an airplane to the given destination
}
```

注意 Airplane::fly 是被如何变成一个 pure virtual function（纯虚拟函数）的。它为飞行提供了 interface（接口）。那个缺省的实现也会出现在 Airplane class 中，但是现在它是一个独立的函数，defaultFly。像 ModelA 和 ModelB 这样需要使用缺省行为的 Classes 只是需要在他们的 fly 的函数体中做一下对 defaultFly 的 inline 调用（但是请参见 Item 30 提供的关于 inline 化和 virtual functions（虚拟函数）的交互作用的信息）：

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...
};
```

对于 ModelC class，不可能在无意中继承到不正确的 fly 的实现，因为 Airplane 中的 pure virtual（纯虚拟）强制要求 ModelC 提供的它自己的 fly 版本。

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...
};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

这一方案并非十分安全（程序员还是能通过 copy-and-paste 使他们自己陷入麻烦），但是它比最初的设计更加可靠。至于 Airplane::defaultFly，它是 protected（保护的）是因为它完全是 Airplane 和它的 derived classes（派生类）的实现细节。使用飞机的客户应该只在意它能飞，而不必管飞行是如何实现的。

Airplane::defaultFly 是一个 non-virtual function（非虚拟函数）这一点也很重要。这是因为 derived class（派生类）不应该重定义这个函数，这是一个在 Item 36 中专门介绍的原则。如果 defaultFly 是 virtual（虚拟的），你就会遇到一个循环的问题：如果某些 derived class（派生类）应该重定义 defaultFly 却忘记了的时候会如何呢？

一些人反对为 interface（接口）和 default implementation（缺省实现）分别提供函数，就像上面的 fly 和 defaultFly 那样。首先，他们注意到，这样做会导致类似的相关函数名污染 class namespace（类名字空间）的问题。然而他们仍然同意 interface（接口）和 default implementation（缺省实现）应该被分开。他们是怎样解决这个表面上的矛盾呢？通过利用以

下事实：pure virtual functions（纯虚拟函数）必须在 concrete derived classes（具体派生类）中被 redeclared（重声明），但是它们也可以有它们自己的实现。以下就是 Airplane hierarchy（继承体系）如何利用这一能力定义一个 pure virtual function（纯虚拟函数）：

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...
};

void Airplane::fly(const Airport& destination)     // an implementation of
{                                                   // a pure virtual function
  default code for flying an airplane to
  the given destination
}

class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

除了用 pure virtual function（纯虚拟函数）Airplane::fly 的函数体代替了独立函数 Airplane::defaultFly 之外，这是一个和前面的几乎完全相同的设计。本质上，fly 可以被拆成两个基本组件。它的 declaration（声明）指定了它的 interface（接口）（这是 derived classes（派生类）必须使用的），而它的 definition（定义）指定它的缺省行为（这是 derived classes（派生类）可以使用的，但只是在他们明确要求这一点时）。将 fly 和 defaultFly 合并，无论如何，你失去了给予这两个函数不同的保护层次的能力：原来是 protected 的代码（通过位于 defaultFly 中实现）现在成为 public（因为它位于 fly 中）。

最后，我们看看 Shape 的 non-virtual function（非虚拟函数），objectID：

```
class Shape {
public:
  int objectID() const;
  ...
};
```

当一个 member function（成员函数）是 non-virtual（非虚拟的）时，不应该指望它在 derived classes（派生类）中的行为会有所不同。实际上，一个 non-virtual member function（非虚拟成员函数）指定了一个 invariant over specialization（超越特殊化的不变量），因为不论一个 derived class（派生类）变得多么特殊，它都把它看作是不允许变化的行为。如下所指除的，

声明一个 non-virtual function（非虚拟函数）的目的是 to have derived classes inherit a function interface as well as a mandatory implementation（使派生类既继承一个函数的接口，又继承一个强制的实现）。

你可以这样考虑 Shape::objectID 的声明，“每一个 Shape object 有一个产生 object identifier（对象标识码），而且这个 object identifier（对象标识码）总是用同样的方法计算出来的，这个方法是由 Shape::objectID 的定义决定的，而且 derived class（派生类）不应该试图改变它的做法。”因为一个 non-virtual function（非虚拟函数）被看作一个 invariant over specialization（超越特殊化的不变量），在 derived class（派生类）中他绝不应该被重定义，细节的讨论参见 Item 36。

对 pure virtual，simple virtual，和 non-virtual functions 的声明的不同允许你精确指定你需要 derived classes（派生类）继承什么东西。分别是 interface only（仅有接口），interface and a default implementation（接口和一个缺省的实现），和 interface and a mandatory implementation（接口和一个强制的实现）。因为这些不同的声明类型意味着根本不同的意义，当你声明你的 member functions（成员函数）时你必须在它们之间仔细地选择。如果你这样做了，你应该可以避免由缺乏经验的类设计者造成的两个最常见的错误。

第一个错误是声明所有的函数为 non-virtual（非虚拟）。这没有给 derived classes（派生类）的特殊化留出空间；non-virtual destructors（非虚拟析构函数）尤其有问题（参见 Item 7）。当然，完全有理由设计一个不作为 base class（基类）使用的类。在这种情况下，一套独享的 non-virtual member functions（非虚拟成员函数）是完全合理的。然而，更通常的情况下，这样的类既可能出于对 virtual（虚拟）和 non-virtual functions（非虚拟函数）之间区别的无知，也可能是对 virtual functions（虚拟函数）的性能成本毫无根据的担心的结果。事实是，几乎任何作为 base class（基类）使用的类都会有 virtual functions（虚拟函数）（还是参见 Item 7）。

如果你关心 virtual functions（虚拟函数）的成本，请允许我提起基于经验的 80-20 规则（参见 Item 30），在一个典型的程序中的情况是，80% 的运行时间花费在执行其中的 20% 的代码上。这个规则是很重要的，因为它意味着，平均下来，你的函数调用中的 80% 可以被虚拟

化而不会对你的程序的整体性能产生哪怕是最轻微的可察觉的影响。在你走进对“你是否能负担得起一个 virtual function（虚拟函数）的成本”忧虑的阴影之前，应该使用一些简单的预防措施，以确保你关注的是你的程序中能产生决定性不同的那 20%。

另一个常见的错误声明所有的 member functions（成员函数）为 virtual（虚拟）。有时候这样做是正确的—— Item 31 的 Interface classes（接口类）可以作为证据。然而，它也可能是缺乏表明态度的决心的类设计者的标志。某些函数在 derived classes（派生类）中不应该被重定义，而且只要在这种情况下，你都应该通过将那些函数声明为 non-virtual（非虚拟）而明确地表达这一点。它不是为那些人服务的，他们假设如果他们只需花一些时间重定义你的所有函数，你的类就会被所有的人用来做所有的事情，如果你有一个 invariant over specialization（超越特殊化的不变量），请直说，不必害怕！

Things to Remember

Inheritance of interface（接口继承）与 inheritance of implementation（实现继承）不同。在 public inheritance（公开继承）下，derived classes（派生类）总是继承 base class interfaces（基类接口）。

Pure virtual functions（纯虚拟函数）指定 inheritance of interface only（仅有接口被继承）。

Simple (impure) virtual functions（简单虚拟函数）指定 inheritance of interface（接口继承）加上 inheritance of a default implementation（缺省实现继承）。

Non-virtual functions（非虚拟函数）指定 inheritance of interface（接口继承）加上 inheritance of a mandatory implementation（强制实现继承）。

Item 35: 考虑可选的 virtual functions（虚拟函数）的替代方法

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

现在你工作在一个视频游戏上，你在游戏中为角色设计了一个 hierarchy（继承体系）。你的游戏中有着变化多端的恶劣环境，角色被伤害或者其它的健康状态降低的情况并不罕见。因此你决定提供一个 member function（成员函数）healthValue，它返回一个象征角色健康状况如何的整数。因为不同的角色计算健康值的方法可能不同，将 healthValue 声明为 virtual（虚拟）似乎是显而易见的设计选择：

```
class GameCharacter {
public:
  virtual int healthValue() const;        // return character's health rating;
  ...                                     // derived classes may redefine this
};
```

healthValue 没有被声明为 pure virtual（纯虚）的事实暗示这里有一个计算健康值的缺省算法（参见 Item 34）。

这确实是一个显而易见的设计选择，而在某种意义上，这是它的缺点。因为这样的设计过于显而易见，你可能不会对它的其它可选方法给予足够的关注。为了帮助你脱离 object-oriented design（面向对象设计）的习惯性道路，我们来考虑一些处理这个问题的其它方法。

The Template Method Pattern via the Non-Virtual Interface Idiom（经由非虚拟接口惯用法实现的模板方法模式）

我们以一个主张 virtual functions（虚拟函数）应该几乎总是为 private（私有的）的有趣观点开始。这一观点的拥护者提出：一个较好的设计应该保留作为 public member function（公有成员函数）的 healthValue，但应将它改为 non-virtual（非虚拟的）并让它调用一个 private virtual function（私有虚拟函数）来做真正的工作，也就是说，doHealthValue：

```
class GameCharacter {
public:
  int healthValue() const               // derived classes do not redefine
  {                                     // this - see Item 36

    ...                                 // do "before" stuff - see below

    int retVal = doHealthValue();       // do the real work

    ...                                 // do "after" stuff - see below

    return retVal;
  }
  ...

private:
  virtual int doHealthValue() const     // derived classes may redefine this
  {
    ...                                 // default algorithm for calculating
  }                                     // character's health
};
```

在这个代码（以及本 Item 的其它代码）中，我在类定义中展示 member functions（成员函数）的本体。就像 Item 30 中所解释的，这会将它们隐式声明为 inline（内联）。我用这种方法展示代码仅仅是这样更易于看到它在做些什么。我所描述的设计与是否 inline 化无关，所以不必深究 member functions（成员函数）定义在类的内部有什么意味深长的含义。根本没有。

这个基本的设计——让客户通过 public non-virtual member functions（公有非虚拟成员函数）调用 private virtual functions（私有虚拟函数）——被称为 non-virtual interface (NVI) idiom（非虚拟接口惯用法）。这是一个更通用的被称为 Template Method（一个模式，很不幸，与 C++ templates（模板）无关）的 design pattern（设计模式）的特殊形式。我将那个 non-virtual function（非虚拟函数）（例如，healthValue）称为 virtual function's wrapper（虚拟函数的外壳）。

NVI idiom（惯用法）的一个优势通过 "do 'before' stuff" 和 "do 'after' stuff" 两个注释在代码中标示出来。这些注释标出的代码片断在做真正的工作的 virtual function（虚拟函数）之前或之后调用。这就意味着那个 wrapper（外壳）可以确保在 virtual function（虚拟函数）被调用前，特定的背景环境被设置，而在调用结束之后，这些背景环境被清理。例如，"before" stuff 可以包括锁闭一个 mutex（互斥体），生成一条日志条目，校验类变量和函数的 preconditions（前提条件）是否被满足，等等。"after" stuff 可以包括解锁一个 mutex（互斥体），校验函数的 postconditions（结束条件），类不变量的恢复，等等。如果你让客户直接调用 virtual functions（虚拟函数），确实没有好的方法能够做到这些。

涉及 derived classes（派生类）重定义 private virtual functions（私有虚拟函数）（这些重定义函数它们不能调用！）的 NVI idiom 可能会搅乱你的头脑。这里没有设计上的矛盾。重定义一个 virtual function（虚拟函数）指定如何做某些事。调用一个 virtual function（虚拟函数）指定什么时候去做。互相之间没有关系。NVI idiom 允许 derived classes（派生类）重定义一个 virtual function（虚拟函数），这样就给了它们控制功能如何实现的能力，但是 base

class（基类）保留了决定函数何时被调用的权利。乍一看很奇怪，但是 C++ 规定 derived classes（派生类）可以重定义 private inherited virtual functions（私有的通过继承得到的函数）是非常明智的。

在 NVI idiom 之下，virtual functions（虚拟函数）成为 private（私有的）并不是绝对必需的。在一些 class hierarchies（类继承体系）中，一个 virtual function（虚拟函数）的 derived class（派生类）实现被期望调用其 base class（基类）的对应物（例如，第 120 页的例子），而为了这样的调用能够合法，虚拟必须成为 protected（保护的），而非 private（私有的）。有时一个 virtual function（虚拟函数）甚至必须是 public（公有的）（例如，polymorphic base classes（多态基类）中的 destructors（析构函数）——参见 Item 7），但这样一来 NVI idiom 就不能被真正应用。

The Strategy Pattern via Function Pointers（经由函数指针实现的策略模式）

NVI idiom 是 public virtual functions（公有虚拟函数）的有趣的可选替代物，但从设计的观点来看，它比装点门也多不了多少东西。毕竟，我们还是在用 virtual functions（虚拟函数）来计算每一个角色的健康值。一个更引人注目的设计主张认为计算一个角色的健康值不依赖于角色的类型——这样的计算根本不需要成为角色的一部分。例如，我们可能需要为每一个角色的 constructor（构造函数）传递一个指向健康值计算函数的指针，而我们可以调用这个函数进行实际的计算：

```
class GameCharacter;                              // forward declaration

// function for the default health calculation algorithm
int defaultHealthCalc(const GameCharacter& gc);

class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);

  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
  : healthFunc(hcf)
  {}

  int healthValue() const
  { return healthFunc(*this); }

  ...

private:
  HealthCalcFunc healthFunc;
};
```

这个方法是另一个通用 design pattern（设计模式）—— Strategy 的简单应用，相对于基于 GameCharacter hierarchy（继承体系）中的 virtual functions（虚拟函数）的方法，它提供了某些更引人注目的机动性：

- 相同角色类型的不同实例可以有不同的健康值计算函数。例如：

```
class EvilBadGuy: public GameCharacter {
public:
 explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
 : GameCharacter(hcf)
 { ... }

 ...

};
int loseHealthQuickly(const GameCharacter&);    // health calculation
int loseHealthSlowly(const GameCharacter&);     // funcs with different
                                               // behavior

EvilBadGuy ebg1(loseHealthQuickly);             // same-type charac-
EvilBadGuy ebg2(loseHealthSlowly);              // ters with different
                                               // health-related
                                               // behavior
```

- 对于一个指定的角色健康值的计算函数可以在运行时改变。例如，GameCharacter 可以提供一个 member function（成员函数）setHealthCalculator，它被允许代替当前的健康值计算函数。

在另一方面，健康值计算函数不再是 GameCharacter hierarchy（继承体系）的一个 member function（成员函数）的事实，意味着它不再拥有访问它所计算的那个对象内部构件的特权。例如，defaultHealthCalc 不能访问 EvilBadGuy 的 non-public（非公有）构件。如果一个角色的健康值计算能够完全基于通过角色的 public interface（公有接口）可以得到的信息，这就没什么问题，但是，如果准确的健康值计算需要 non-public（非公有）信息，就会有问题。实际上，在任何一个你要用 class（类）外部的等价机能（例如，经由一个 non-member non-friend function（非成员非友元函数）或经由另一个 class（类）的 non-friend member function（非友元成员函数））代替 class（类）内部的机能（例如，经由一个 member function（成员函数））的时候，它都是一个潜在的问题。这个问题将持续影响本 Item 的剩余部分，因为所有我们要考虑的其它设计选择都包括 GameCharacter hierarchy（继承体系）的外部函数的使用。

作为一个通用规则，解决对“non-member functions（非成员函数）对类的 non-public（非公有）构件的访问的需要”的唯一方法就是削弱类的 encapsulation（封装性）。例如，class（类）可以将 non-member functions（非成员函数）声明为 friends（友元），或者，它可以提供对“在其它情况下它更希望保持隐藏的本身的实现部分”的 public accessor functions（公有访问者函数）。使用一个 function pointer（函数指针）代替一个 virtual function（虚拟函数）的优势（例如，具有逐对象健康值计算函数的能力和在运行时改变这样的函数的能力）是否能抵消可能的降低 GameCharacter 的 encapsulation（封装性）的需要是你必须在设计时就做出决定的重要部分。

The Strategy Pattern via tr1::function（经由 tr1::function 实现的策略模式）

一旦你习惯了 templates（模板）和 implicit interfaces（隐式接口）（参见 Item 41）的应用，function-pointer-based（基于函数指针）的方法看上去就有些死板了。健康值的计算为什么必须是一个 function（函数），而不能是某种简单的行为类似 function（函数）的东西

（例如，一个 function object（函数对象））？如果它必须是一个 function（函数），为什么不能是一个 member function（成员函数）？为什么它必须返回一个 int，而不是某种能够转型为 int 的类型？

如果我们用一个 tr1::function 类型的对象代替一个 function pointer（函数指针）（诸如 healthFunc），这些约束就会消失。就像 Item 54 中的解释，这样的对象可以持有 any callable entity（任何可调用实体）（例如，function pointer（函数指针），function object（函数对象），或 member function pointer（成员函数指针）），这些实体的标志性特征就是兼容于它所期待的东西。我们马上就会看到这样的设计，这次使用了 tr1::function：

```
class GameCharacter;                                 // as before
int defaultHealthCalc(const GameCharacter& gc);      // as before

class GameCharacter {
public:
   // HealthCalcFunc is any callable entity that can be called with
   // anything compatible with a GameCharacter and that returns anything
   // compatible with an int; see below for details
   typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
   explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
   : healthFunc(hcf)
   {}

   int healthValue() const
   { return healthFunc(*this);   }

   ...

private:
  HealthCalcFunc healthFunc;
};
```

就像你看到的，HealthCalcFunc 是一个 tr1::function instantiation（实例化）的 typedef。这意味着它的行为类似一个普通的 function pointer（函数指针）类型。我们近距离看看 HealthCalcFunc 究竟是一个什么东西的 typedef：

```
std::tr1::function<int (const GameCharacter&)>
```

这里我突出了这个 tr1::function instantiation（实例化）的“target signature（目标识别特征）”。这个 target signature（目标识别特征）是“取得一个引向 const GameCharacter 的 reference（引用），并返回一个 int 的函数”。这个 tr1::function 类型的（例如，HealthCalcFunc 类型的）对象可以持有兼容于这个 target signature（目标识别特征）的 any callable entity（任何可调用实体）。兼容意味着这个实体的参数能够隐式地转型为一个 const GameCharacter&，而它的返回类型能够隐式地转型为一个 int。

与我们看到的最近一个设计（在那里 GameCharacter 持有一个指向一个函数的指针）相比，这个设计几乎相同。仅有的区别是目前的 GameCharacter 持有一个 tr1::function 对象——指向一个函数的 generalized（泛型化）指针。除了达到“clients（客户）在指定健康值计算函数时有更大的灵活性”的效果之外，这个变化是如此之小，以至于我宁愿对它视而不见：

```
short calcHealth(const GameCharacter&);          // health calculation
                                                 // function; note
                                                 // non-int return type

struct HealthCalculator {                        // class for health
  int operator()(const GameCharacter&) const     // calculation function
  { ... }                                        // objects
};

class GameLevel {
public:
  float health(const GameCharacter&) const;      // health calculation
  ...                                            // mem function; note
};                                               // non-int return type


class EvilBadGuy: public GameCharacter {         // as before
  ...
};
class EyeCandyCharacter:   public GameCharacter {  // another character
  ...                                              // type; assume same
};                                                 // constructor as
                                                   // EvilBadGuy


EvilBadGuy ebg1(calcHealth);                       // character using a
                                                   // health calculation
                                                   // function


EyeCandyCharacter ecc1(HealthCalculator());        // character using a
                                                   // health calculation
                                                   // function object

GameLevel currentLevel;
...
EvilBadGuy ebg2(                                   // character using a
  std::tr1::bind(&GameLevel::health,               // health calculation
          currentLevel,                            // member function;
          _1)                                      // see below for details
);
```

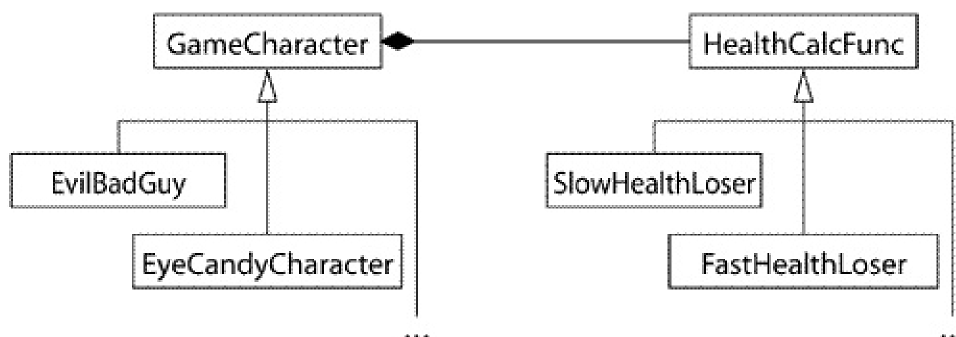
就个人感觉而言：我发现 tr1::function 能让你做的事情是如此让人惊喜，它令我浑身兴奋异常。如果你没有感到兴奋，那可能是因为你正目不转睛地盯着 ebg2 的定义并对 tr1::bind 的调用会发生什么迷惑不解。请耐心地听我解释。

比方说我们要计算 ebg2 的健康等级，应该使用 GameLevel class（类）中的 health member function（成员函数）。现在，GameLevel::health 是一个被声明为取得一个参数（一个引向 GameCharacter 的引用）的函数，但是它实际上取得了两个参数，因为它同时得到一个隐式的 GameLevel 参数——指向 this。然而，GameCharacters 的健康值计算函数只取得单一的参数：将被计算健康值的 GameCharacter。如果我们要使用 GameLevel::health 计算 ebg2 的健康值，我们必须以某种方式“改造”它，以使它适应只取得唯一的参数（一个 GameCharacter），而不是两个（一个 GameCharacter 和一个 GameLevel）。在本例中，我们总是要使用 currentLevel 作为 GameLevel 对象来计算 ebg2 的健康值，所以每次调用 GameLevel::health 计算 ebg2 的健康值时，我们就要 "bind"（凝固）currentLevel 来作为 GameLevel 的对象来使用。这就是 tr1::bind 的调用所做的事情：它指定 ebg2 的健康值计算函数应该总是使用 currentLevel 作为 GameLevel 对象。

我们跳过一大堆的细节，诸如为什么 "_1" 意味着“当为了 ebg2 调用 GameLevel::health 时使用 currentLevel 作为 GameLevel 对象”。这样的细节并没有什么启发性，而且它们将转移我所关注的基本点：在计算一个角色的健康值时，通过使用 tr1::function 代替一个 function pointer（函数指针），我们将允许客户使用 any compatible callable entity（任何兼容的可调用实体）。很酷是不是？

The "Classic" Strategy Pattern（“经典的”策略模式）

如果你比 C++ 更加深入地进入 design patterns（设计模式），一个 Strategy 的更加习以为常的做法是将 health-calculation function（健康值计算函数）做成一个独立的 health-calculation hierarchy（健康值计算继承体系）的 virtual member function（虚拟成员函数）。做成的 hierarchy（继承体系）设计看起来就像这样：



如果你不熟悉 UML 记法，这不过是在表示当把 EvilBadGuy 和 EyeCandyCharacter 作为 derived classes（派生类）时，GameCharacter 是这个 inheritance hierarchy（继承体系）的根；HealthCalcFunc 是另一个带有 derived classes（派生类）SlowHealthLoser 和 FastHealthLoser 的 inheritance hierarchy（继承体系）的根；而每一个 GameCharacter 类型的对象包含一个指向“从 HealthCalcFunc 派生的对象”的指针。

这就是相应的框架代码：

```
class GameCharacter;                            // forward declaration

class HealthCalcFunc {
public:

  ...
  virtual int calc(const GameCharacter& gc) const
  { ... }
  ...

};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
  explicit GameCharacter(HealthCalcFunc *phcf = &defaultHealthCalc)
  : pHealthCalc(phcf)
  {}

  int healthValue() const
  { return pHealthCalc->calc(*this);}

  ...

private:
  HealthCalcFunc *pHealthCalc;
};
```

这个方法的吸引力在于对于熟悉"标准的"Strategy pattern（策略模式）实现的人可以很快地识别出来，再加上它提供了通过在 HealthCalcFunc hierarchy（继承体系）中增加一个 derived class（派生类）而微调已存在的健康值计算算法的可能性。

Summary（概要）

这个 Item 的基本建议是当你为尝试解决的问题寻求一个设计时，你应该考虑可选的 virtual functions（虚拟函数）的替代方法。以下是对我们考察过的可选方法的一个简略的回顾：

- 使用 non-virtual interface idiom (NVI idiom)（非虚拟接口惯用法），这是用 public non-virtual member functions（公有非虚拟成员函数）包装可访问权限较小的 virtual functions（虚拟函数）的 Template Method design pattern（模板方法模式）的一种形式。
- 用 function pointer data members（函数指针数据成员）代替 virtual functions（虚拟函数），一种 Strategy design pattern（策略模式）的显而易见的形式。
- 用 tr1::function data members（数据成员）代替 virtual functions（虚拟函数），这样就允许使用兼容于你所需要的东西的 any callable entity（任何可调用实体）。这也是 Strategy design pattern（策略模式）的一种形式。
- 用 virtual functions in another hierarchy（另外一个继承体系中的虚拟函数）代替 virtual functions in one hierarchy（单独一个继承体系中的虚拟函数）。这是 Strategy design pattern（策略模式）的习以为常的实现。

这不是一个可选的 virtual functions（虚拟函数）的替代设计的详尽无遗的列表，但是它足以使你确信这些是可选的方法。此外，它们之间互为比较的优劣应该使你考虑它们时更为明确。

为了避免陷入 object-oriented design（面向对象设计）的习惯性道路，时不时地给车轮一些有益的颠簸。有很多其它的道路。值得花一些时间去考虑它们。

Things to Remember

可选的 virtual functions（虚拟函数）的替代方法包括 NVI 惯用法和 Strategy design pattern（策略模式）的各种变化形式。NVI 惯用法本身是 Template Method design pattern（模板方法模式）的一个实例。

将一个机能从一个 member function（成员函数）中移到 class（类）之外的某个函数中的一个危害是 non-member function（非成员函数）没有访问类的 non-public members（非公有成员）的途径。

tr1::function 对象的行为类似 generalized function pointers（泛型化的函数指针）。这样的对象支持所有兼容于一个给定的目标特征的 callable entities（可调用实体）。

Item 36: 绝不要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我告诉你 class（类）D 从 class（类）B publicly derived（公有继承），而且在 class（类）B 中定义了一个 public member function（公有成员函数）mf。mf 的参数和返回值类型是无关紧要的，所以我们就假设它们都是 void。换句话说，我的意思是：

```
class B {
public:
  void mf();
  ...
};
class D: public B { ... };
```

甚至不必知道关于 B，D，或 mf 的任何事情，给定一个类型为 D 的 object（对象）x，

```
D x; // x is an object of type D
```

对此你或许非常吃惊，

```
B *pB = &x; // get pointer to x

pB->mf(); // call mf through pointer
```

的行为不同于以下代码：

```
D *pD = &x; // get pointer to x

pD->mf(); // call mf through pointer
```

因为在两种情况中，你都调用了 object（对象）x 中的 member function（成员函数）mf。因为两种情况中都是同样的 function（函数）和同样的 object（对象），它们的行为应该有相同的方式，对吗？

是的，应该。但是也可能不，特别地，如果 mf 是 non-virtual（非虚拟）而 D 定义了它自己的版本的 mf：

```
class D: public B {
public:
  void mf();                        // hides B::mf; see Item33

  ...

};

pB->mf();                           // calls B::mf

pD->mf();                           // calls D::mf
```

这种行为两面性的原因是像 B::mf 和 D::mf 这样的 non-virtual functions（非虚拟函数）是 statically bound（静态绑定）的（参见 Item 37）。这就意味着因为 pB 被声明为 pointer-to-B 类型，所以，即使就像本例中的做法，让 pB 指向一个从 B 继承的类的对象，通过 pB 调用的 non-virtual functions（非虚拟函数）也总是定义在 class B 中的那一个。

在另一方面，virtual functions（虚拟函数）是 dynamically bound（动态绑定）的（再次参见 Item 37），所以它们不会发生这个问题。如果 mf 是一个 virtual function（虚拟函数），无论通过 pB 还是 pD 调用 mf 都将导致 D::mf 的调用，因为 pB 和 pD 都实际地指向一个 type（类型）D 的 object（对象）。

如果你在编写 class D 而且你重定义了一个你从 class B 继承到的 non-virtual function（非虚拟函数）mf，D 的 objects（对象）将很可能表现出不协调的行为。特别是，当 mf 被调用时，任何给定的 D object（对象）的行为既可能像 B 也可能像 D，而且决定因素与 object（对象）本身无关，但是和指向它的 pointer（指针）的声明类型有关。references（引用）也会像 pointers（指针）一样表现出莫名其妙的行为。

但这仅仅是一个从实用出发的论据。我知道，你真正需要的是不能重定义 inherited non-virtual functions（通过继承得到的非虚拟函数）的理论上的理由。我很愿意效劳。

Item 32 解释了 public inheritance（公有继承）意味着 is-a，Item 34 记述了为什么在一个 class（类）中声明一个 non-virtual function（非虚拟函数）是为这个 class（类）设定一个 invariant over specialization（超越特殊化的不变量），如果你将这些经验应用于 classes（类）B 和 D 以及 non-virtual member function（非虚拟函数）B::mf，那么：

每一件适用于 B objects（对象）的事情也适用于 D objects（对象），因为每一个 D objects 都 is-a（是一个）D objects（对象）；

从 B 继承的 classes（类）必须同时继承 mf 的 interface（接口）和 implementation（实现），因为 mf 在 B 中是 non-virtual（非虚拟）的。

现在，如果 D 重定义 mf，你的设计中就有了一处矛盾。如果 D 真的需要实现不同于 B 的 mf，而且如果每一个 B objects（对象）——无论如何特殊——都必须使用 B 对 mf 的实现，那么每一个 D 都 is-a（是一个）B 就完全不成立。在那种情况下，D 就不应该从 B publicly inherit（公有继承）。另一方面，如果 D 真的必须从 B publicly inherit（公有继承），而且如果 D 真的需要实现不同于 B 的 mf，那么 mf 反映了一个 B 的 invariant over specialization（超越特殊化的不变量）就不会成立。在那种情况下，mf 应该是 virtual（虚

拟）的。最后，如果每一个 D 真的都 is-a（是一个）B，而且如果 mf 真的相当于一个 B 的 invariant over specialization（超越特殊化的不变量），那么 D 就不会真的需要重定义 mf，而且想都不能想。

不管使用那一条规则，必须做出某些让步，而且无条件地禁止重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）。

如果阅读这个 Item 给你 déjà vu（似曾相识）的感觉，那可能是因为你已经阅读了 Item 7，那个 Item 解释了为什么 polymorphic base classes（多态基类）中的 destructors（析构函数）应该是 virtual（虚拟）的。如果你违反了那个 guideline（指导方针）（例如，如果你在一个 polymorphic base class（多态基类）中声明一个 non-virtual destructor（非虚拟析构函数）），你也同时违反了这里这个 guideline（指导方针），因为 derived classes（派生类）总是要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）：base class（基类）的 destructor（析构函数）。甚至对于没有声明 destructor（析构函数）的 derived classes（派生类）这也是成立的，因为，就像 Item 5 的解释，destructor（析构函数）是一个“如果你没有定义你自己的，编译器就会为你生成一个”的 member functions（成员函数）。其实，Item 7 只相当于本 Item 的一个特殊情况，尽管它重要到足以把它提出来独立成篇。

Things to Remember

绝不要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）。

Item 37: 绝不要重定义一个函数的 inherited default parameter value（通过继承得到的缺省参数值）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我们直接着手简化这个话题。只有两种函数能被你 inherit（继承）：virtual（虚拟的）和 non-virtual（非虚拟的）。然而，重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）永远都是一个错误（参见 Item 36），所以我们可以安全地将我们的讨论限制在你继承了一个 virtual function with a default parameter value（带有一个缺省参数值的虚拟函数）的情形。

在这种情况下，本 Item 的理由就变得非常地直截了当：virtual functions（虚拟函数）是 dynamically bound（动态绑定），而 default parameter values（缺省参数值）是 statically bound（静态绑定）。

那又怎样呢？你说 static（静态）和 dynamic binding（动态绑定）之间的区别早已塞入你负担过重的头脑？（不要忘了，static binding（静态绑定）也以 early binding（前期绑定）闻名，而 dynamic binding（动态绑定）也以 late binding（后期绑定）闻名。）那么，我们就再来回顾一下。

一个 object（对象）的 static type（静态类型）就是你在程序文本中声明给它的 type（类型）。考虑这个 class hierarchy（类继承体系）：

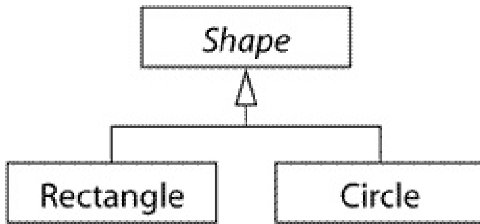
```
// a class for geometric shapes
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  // all shapes must offer a function to draw themselves
  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  // notice the different default parameter value — bad!
  virtual void draw(ShapeColor color = Green) const;
  ...
};

class Circle: public Shape {
public:
  virtual void draw(ShapeColor color) const;
  ...
};
```

直观地看，它看起来就像这个样子：



现在考虑这些 pointers（指针）：

```
Shape *ps;                        // static type = Shape*
Shape *pc = new Circle;           // static type = Shape*
Shape *pr = new Rectangle;        // static type = Shape*
```

在本例中，ps，pc 和 pr 全被声明为 pointer-to-Shape 类型，所以它们全都以此作为它们的 static type（静态类型）。注意这就使得它们真正指向的东西完全没有区别——无论如何，它们的 static type（静态类型）都是 Shape*。

一个 object（对象）的 dynamic type（动态类型）取决于它当前引用的 object（对象）的 type（类型）。也就是说，它的 dynamic type（动态类型）表明它有怎样的行为。在上面的例子中，pc 的 dynamic type（动态类型）是 Circle*，而 pr 的 dynamic type（动态类型）是 Rectangle*。至于 ps，它没有一个实际的 dynamic type（动态类型），因为它（还）不能引用任何 object（对象）。

dynamic types（动态类型），就像它的名字所暗示的，能在程序运行中变化，特别是通过 assignments（赋值）：

```
ps = pc;                        // ps's dynamic type is
                                // now Circle*

ps = pr;                        // ps's dynamic type is
                                // now Rectangle*
```

virtual functions（虚拟函数）是 dynamically bound（动态绑定），意味着被调用的特定函数取决于被用来调用它的那个 object（对象）的 dynamic type（动态类型）：

```
pc->draw(Shape::Red);             // calls Circle::draw(Shape::Red)

pr->draw(Shape::Red);             // calls Rectangle::draw(Shape::Red)
```

我知道，这全是老生常谈；你的确已经理解了 virtual functions（虚拟函数）。但是，当你考虑 virtual functions with default parameter values（带有缺省参数值的虚拟函数）时，就全乱了套，因为，如上所述，virtual functions（虚拟函数）是 dynamically bound（动态绑定），但 default parameters（缺省参数）是 statically bound（静态绑定）。这就意味着你最终调用了一个定义在 derived class（派生类）中的 virtual function（虚拟函数）却使用了一个来自 base class（基类）的 default parameter value（缺省参数值）。

```
pr->draw(); // calls Rectangle::draw(Shape::Red)!
```

在此情况下，pr 的 dynamic type（动态类型）是 Rectangle*，所以正像你所希望的，Rectangle 的 virtual function（虚拟函数）被调用。在 Rectangle::draw 中，default parameter value（缺省参数值）是 Green。然而，因为 pr 的 static type（静态类型）是 Shape*，这个函数调用的 default parameter value（缺省参数值）是从 Shape class 中取得的，而不是 Rectangle class！导致的结果就是一个调用由“奇怪的和几乎完全出乎意料的 Shape 和 Rectangle 两个 classes（类）中的 draw 声明的混合物”所组成。

ps，pc，和 pr 是 pointers（指针）的事实与这个问题并无因果关系，如果它们是 references（引用），问题依然会存在。唯一重要的事情是 draw 是一个 virtual function（虚拟函数），而它的一个 default parameter values（缺省参数值）在一个 derived class（派生类）中被重定义。

为什么 C++ 要坚持按照这种不正常的方式动作？答案是为了运行时效率。如果 default parameter values（缺省参数值）是 dynamically bound（动态绑定），compilers（编译器）就必须提供一种方法在运行时确定 virtual functions（虚拟函数）的 parameters（参数）的 default value(s)（缺省值），这比目前在编译期确定它们的机制更慢而且更复杂。最终的决定偏向了速度和实现的简单这一边，而造成的结果就是你现在可以享受高效运行的乐趣，但是，如果你忘记留心本 Item 的建议，就会陷入困惑。

这样就很彻底而且完美了，但是看看如果你试图遵循本规则为 base（基类）和 derived classes（派生类）的用户提供同样的 default parameter values（缺省参数值）时会发生什么：

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  virtual void draw(ShapeColor color = Red) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  virtual void draw(ShapeColor color = Red) const;
  ...
};
```

噢，code duplication（代码重复）。code duplication（代码重复）带来 dependencies（依赖关系）：如果 Shape 中的 default parameter values（缺省参数值）发生变化，所有重复了它的 derived classes（派生类）必须同时变化。否则它们就陷入重定义一个 inherited default parameter value（通过继承得到的缺省参数值）。怎么办呢？

当你要一个 virtual function（虚拟函数）按照你希望的方式运行有困难的时候，考虑可选的替代设计是很明智的，而且 Item 35 给出了多个 virtual function（虚拟函数）的替代方法。替代方法之一是 non-virtual interface idiom (NVI idiom)（非虚拟接口惯用法）：用 base

class（基类）中的 public non-virtual function（公有非虚拟函数）调用 derived classes（派生类）可能重定义的 private virtual function（私有虚拟函数）。这里，我们用 non-virtual function（非虚拟函数）指定 default parameter（缺省参数），同时使用 virtual function（虚拟函数）做实际的工作：

```
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  void draw(ShapeColor color = Red) const           // now non-virtual
  {
    doDraw(color);                                  // calls a virtual
  }

  ...

private:
  virtual void doDraw(ShapeColor color) const = 0;  // the actual work is
};                                                  // done in this func

class Rectangle: public Shape {
public:

  ...

private:
  virtual void doDraw(ShapeColor color) const;       // note lack of a
  ...                                                // default param val.
};
```

因为 non-virtual functions（非虚拟函数）绝不应该被 derived classes（派生类）overridden（覆盖）（参见 Item 36），这个设计使得 draw 的 color parameter（参数）的 default value（缺省值）应该永远是 Red 变得明确。

Things to Remember

绝不要重定义一个 inherited default parameter value（通过继承得到的缺省参数值），因为 default parameter value（缺省参数值）是 statically bound（静态绑定），而 virtual functions——应该是你可以 overriding（覆盖）的仅有的函数——是 dynamically bound（动态绑定）。

Item 38: 通过 composition（复合）模拟 "has-a"（有一个）或 "is-implemented-in-terms-of"（是根据……实现的）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

composition（复合）是在 objects of one type（一个类型的对象）包含 objects of another type（另一个类型的对象）时，types（类型）之间的关系。例如：

```
class Address { ... };             // where someone lives

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  std::string name;               // composed object
  Address address;                // ditto
  PhoneNumber voiceNumber;        // ditto
  PhoneNumber faxNumber;          // ditto
};
```

此例之中，Person objects（对象）由 string，Address，和 PhoneNumber objects（对象）组成。在程序员中，术语 composition（复合）有很多同义词。它也可以称为 layering，containment，aggregation，和 embedding。

Item 32 解释了 public inheritance（公有继承）意味着 "is-a"（是一个）。composition（复合）也有一个含意。实际上，他有两个含意。composition（复合）既意味着 "has-a"（有一个），又意味着 "is-implemented-in-terms-of"（是根据……实现的）。这是因为你要在你的软件中处理两个不同的 domains（领域）。你程序中的一些 objects（对象）对应你所模拟的世界里的东西，例如，people（人），vehicles（交通工具），video frames（视频画面）等等。这样的 objects（对象）是 application domain（应用领域）的部分。另外的 objects（对象）纯粹是 implementation artifacts（实现的产物），例如，buffers（缓冲区），mutexes（互斥体），search trees（搜索树）等等。这些各类 objects（对象）定义应你的软件的 implementation domain（实现领域）。当 composition（复合）发生在 application domain（应用领域）的 objects（对象）之间，它表达一个 has-a（有一个）的关系，当它发生在 implementation domain（实现领域），它表达一个 is-implemented-in-terms-of（是根据……实现的）的关系

上面的 Person class（类）示范了 has-a（有一个）的关系。一个 Person object（对象）has a（有一个）名字，一个地址，以及语音和传真电话号码。你不能说一个人 is a（是一个）名字或一个人 is an（是一个）地址。你可以说一个人 has a（有一个）名字和 has an（有一个）地址。大多数人对此区别不难理解，所以混淆 is-a（是一个）和 has-a（有一个）之间的角色的情况非常少见。

is-a（是一个）和 is-implemented-in-terms-of（是根据……实现的）之间的区别稍微有些棘手。例如，假设你需要一个类的模板来表现相当小的 objects（对象）的 sets，也就是说，排除重复的集合。因为 reuse（复用）是一件受人欢迎的事情，你的第一个直觉就是使用标准库中的 set template（模板）。当你能使用已经被写好的东西时，为什么还要写一个新的 template（模板）呢？

不幸的是，set 的典型实现导致每个元素三个指针的开销。这是因为 sets 通常被作为 balanced search trees（平衡搜索树）来实现，这允许它们保证 logarithmic-time（对数时间）的 lookups（查找），insertions（插入）和 erasures（删除）。当速度比空间更重要的时候，这是一个合理的设计，但是当空间比速度更重要时，对你的程序来说就有问题了。因而，对你来说，标准库的 set 为你提供了不合理的交易。看起来你终究还是要写你自己的 template（模板）。

reuse（复用）依然是一件受人欢迎的事情。作为 data structure（数据结构）的专家，你知道实现 sets 的诸多选择，其中一种是使用 linked lists（线性链表）。你也知道标准的 C++ 库中有一个 list template（模板），所以你决定（复）用它。

具体地说，你决定让你的新的 Set template（模板）从 list 继承。也就是说，Set<T> 将从 list<T> 继承。毕竟，在你的实现中，一个 Set object（对象）实际上就是一个 list object（对象）。于是，你就像这样声明你的 Set template（模板）：

```
template<typename T>                       // the wrong way to use list for Set
class Set: public std::list<T> { ... };
```

在这里，看起来每件事情都很好。但实际上有一个很大的错误。就像 Item 32 中的解释，如果 D is-a（是一个）B，对于 B 成立的每一件事情对 D 也成立。然而，一个 list object（对象）可以包含重复，所以如果值 3051 被插入一个 list<int> 两次，那个 list 将包含 3051 的两个拷贝。与此对照，一个 Set 不可以包含重复，所以如果值 3051 被插入一个 Set<int> 两次，那个 set 只包含该值的一个拷贝。因此一个 Set is-a（是一个）list 是不正确的，因为对 list objects（对象）成立的某些事情对 Set objects（对象）不成立。

因为这两个 classes（类）之间的关系不是 is-a（是一个），public inheritance（公有继承）不是模拟这个关系的正确方法。正确的方法是认识到一个 Set object（对象）可以 be implemented in terms of a list object（是根据一个 list 对象实现的）：

```
template<class T>                    // the right way to use list for Set
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  std::size_t size() const;

private:
  std::list<T> rep;                  // representation for Set data
};
```

Set 的 member functions（成员函数）可以极大程度地依赖 list 和标准库的其它部分已经提供的机能，所以只要你熟悉了用 STL 编程的基本方法，实现就非常简单了：

```
template<typename T>
bool Set<T>::member(const T& item) const
{
  return std::find(rep.begin(), rep.end(), item) != rep.end();
}

template<typename T>
void Set<T>::insert(const T& item)
{
  if (!member(item)) rep.push_back(item);
}

template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =               // see Item 42 for info on
    std::find(rep.begin(), rep.end(), item);         // "typename" here
  if (it != rep.end()) rep.erase(it);
}

template<typename T>
std::size_t Set<T>::size() const
{
  return rep.size();
}
```

这些函数足够简单，使它们成为 inlining（内联化）的合理候选者，可是我知道在坚定 inlining（内联化）的决心之前，你可能需要回顾一下 Item 30 中的讨论。

一个有说服力的观点是，根据 Item 18 的关于将 interfaces（接口）设计得易于正确使用，而难以错误使用的论述，如果要遵循 STL container（容器）的惯例，Set 的 interface（接口）应该更多，但是在这里遵循那些惯例就需要在 Set 中填充大量 stuff（材料），这将使得它和 list 之间的关系变得暧昧不清。因为这个关系是本 Item 的重点，我们用教学的清晰性替换了 STL 的兼容性。除此之外，Set 的 interface（接口）的幼稚不应该遮掩关于 Set 的无可争辩的正确：它和 list 之间的关系。这个关系不是 is-a（是一个）（虽然最初看上去可能很像），而是 is-implemented-in-terms-of（是根据……实现的）。

Things to Remember

- composition（复合）与 public inheritance（公有继承）的意义完全不同。

* 在 application domain（应用领域）中，composition（复合）意味着 has-a（有一个）。在 implementation domain（实现领域）中意味着 is-implemented-in-terms-of（是根据……实现的）。

Item 39: 谨慎使用 private inheritance（私有继承）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 32 论述了 C++ 将 public inheritance（公有继承）视为一个 is-a 关系。当给定一个 hierarchy（继承体系），其中有一个 class Student 从一个 class Person 公有继承，当为了成功调用一个函数而必需时，就要将 Students 隐式转型为 Persons，它通过向编译器展示来做到这一点。用 private inheritance（私有继承）代替 public inheritance（公有继承）把这个例子的一部分重做一下是值得的：

```
class Person { ... };
class Student: private Person { ... };     // inheritance is now private

void eat(const Person& p);                 // anyone can eat

void study(const Student& s);              // only students study

Person p;                                  // p is a Person
Student s;                                 // s is a Student

eat(p);                                    // fine, p is a Person

eat(s);                                    // error! a Student isn't a Person
```

很明显，private inheritance（私有继承）不意味着 is-a。那么它意味着什么呢？

“喂！”你说：“在我们得到它的含义之前，我们先看看它的行为。private inheritance（私有继承）有怎样的行为呢？”好吧，支配 private inheritance（私有继承）的第一个规则你只能从动作中看到：与 public inheritance（公有继承）对照，如果 classes（类）之间的 inheritance relationship（继承关系）是 private（私有）的，编译器通常不会将一个 derived class object（派生类对象）（诸如 Student）转型为一个 base class object（基类对象）（诸如 Person）。这就是为什么为 object（对象）s 调用 eat 会失败。第二个规则是从一个 private base class（私有基类）继承的 members（成员）会成为 derived class（派生类）的 private members（私有成员），即使它们在 base class（基类）中是 protected（保护）的或 public（公有）的。

行为不过如此。这就给我们带来了含义。private inheritance（私有继承）意味着 is-implemented-in-terms-of（是根据……实现的）。如果你使 class（类）D 从 class（类）B 私有继承，你这样做是因为你对于利用在 class（类）B 中才可用的某些特性感兴趣，而不是因为在 types（类型）B 和 types（类型）D 的 objects（对象）之间有什么概念上的关系。同样地，private inheritance（私有继承）纯粹是一种实现技术。（这也就是为什么你从一个

private base class（私有基类）继承的每一件东西都在你的 class（类）中变成 private（私有）的原因：它全部都是实现的细节。）利用 Item 34 中提出的条款，private inheritance（私有继承）意味着只有 implementation（实现）应该被继承；interface（接口）应该被忽略。如果 D 从 B 私有继承，它就意味着 D objects are implemented in terms of B objects（D 对象是根据 B 对象实现的），没有更多了。private inheritance（私有继承）在 software design（软件设计）期间没有任何意义，只在 software implementation（软件实现）期间才有。

private inheritance（私有继承）意味着 is-implemented-in-terms-of（是根据……实现的）的事实有一点混乱，因为 Item 38 指出 composition（复合）也有同样的含义。你怎么预先在它们之间做出选择呢？答案很简单：只要你能就用 composition（复合），只有在绝对必要的时候才用 private inheritance（私有继承）。什么时候是绝对必要呢？主要是当 protected members（保护成员）和/或 virtual functions（虚拟函数）掺和进来的时候，另外还有一种与空间相关的极端情况会使天平向 private inheritance（私有继承）倾斜。我们稍后再来操心这种极端情况。毕竟，它只是一种极端情况。

假设我们工作在一个包含 Widgets 的应用程序上，而且我们认为我们需要更好地理解 Widgets 是怎样被使用的。例如，我们不仅要知道 Widget member functions（成员函数）被调用的频度，还要知道 call ratios（调用率）随着时间的流逝如何变化。带有清晰的执行阶段的程序在不同的执行阶段可以有不同的行为侧重。例如，一个编译器在解析阶段对函数的使用与优化和代码生成阶段就有很大的不同。

我们决定修改 Widget class 以持续跟踪每一个 member function（成员函数）被调用了多少次。在运行时，我们可以周期性地检查这一信息，与每一个 Widget 的这个值相伴的可能还有我们觉得有用的其它数据。为了进行这项工作，我们需要设立某种类型的 timer（计时器），以便在到达收集用法统计的时间时我们可以知道。

尽可能复用已有代码，而不是写新的代码，我在我的工具包中翻箱倒柜，而且满意地找到下面这个 class（类）：

```
class Timer {
public:
  explicit Timer(int tickFrequency);
   virtual void onTick() const;          // automatically called for each tick
  ...
};
```

这正是我们要找的：一个我们能够根据我们的需要设定 tick 频率的 Timer object，而在每次 tick 时，它调用一个 virtual function（虚拟函数）。我们可以重定义这个 virtual function（虚拟函数）以便让它检查 Widget 所在的当前状态。很完美！

为了给 Widget 重定义 Timer 中的一个 virtual function（虚拟函数），Widget 必须从 Timer 继承。但是 public inheritance（公有继承）在这种情况下不合适。Widget is-a（是一个）Timer 不成立。Widget 的客户不应该能够在一个 Widget 上调用 onTick，因为在概念上那不是的

Widget 的 interface（接口）的一部分。允许这样的函数调用将使客户更容易误用 Widget 的 interface（接口），这是一个对 Item 18 的关于“使接口易于正确使用，而难以错误使用”的建议的明显违背。public inheritance（公有继承）在这里不是正确的选项。

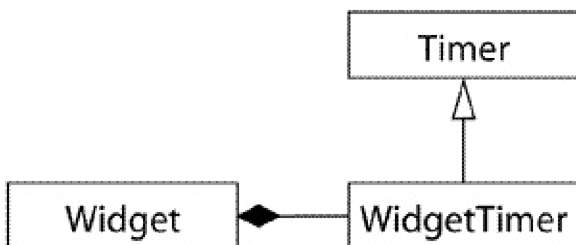
因此我们就 inherit privately（秘密地继承）：

```
class Widget: private Timer {
private:
  virtual void onTick() const;           // look at Widget usage data, etc.
  ...
};
```

通过 private inheritance（私有继承）的能力，Timer 的 public（公有）onTick 函数在 Widget 中变成 private（私有）的，而且在我们重新声明它的时候，也把它保留在那里。重复一次，将 onTick 放入 public interface（公有接口）将误导客户认为他们可以调用它，而这违背了 Item 18。

这是一个很好的设计，但值得一提的是，private inheritance（私有继承）并不是绝对必要的。如果我们决定用 composition（复合）来代替，也是可以的。我们仅需要在我们从 Timer 公有继承来的 Widget 内声明一个 private nested class（私有嵌套类），在那里重定义 onTick，并在 Widget 中放置一个那个类型的 object（对象）。以下就是这个方法的概要：

```
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
    virtual void onTick() const;
    ...
  };
   WidgetTimer timer;
  ...
};
```



这个设计比只用了 private inheritance（私有继承）的那一个更复杂，因为它包括 (public) inheritance（（公有）继承）和 composition（复合）两者，以及一个新 class (WidgetTimer) 的引入。老实说，我出示它主要是为了提醒你有多于一条的道路通向一个设计问题，而且它对于考虑多种方法的自我训练也有相当的价值（参见 Item 35）。然而，我可以想到为什么你可能更愿意用 public inheritance（公有继承）加 composition（复合）而不用 private inheritance（私有继承）的两个原因。

首先，你可能要做出允许 Widget 有 derived classes（派生类）的设计，但是你还可能要禁止 derived classes（派生类）重定义 onTick。如果 Widget 从 Timer 继承，那是不可能的，即使 inheritance（继承）是 private（私有）的也不行。（回忆 Item 35 derived classes（派生类）可以重定义 virtual functions（虚拟函数），即使调用它们是不被允许的。）但是如果 WidgetTimer 在 Widget 中是 private（私有）的而且是从 Timer 继承的，Widget 的 derived classes（派生类）就不能访问 WidgetTimer，因此就不能从它继承或重定义它的 virtual functions（虚拟函数）。如果你曾在 Java 或 C# 中编程并且错过了禁止 derived classes（派生类）重定义 virtual functions（虚拟函数）的能力（也就是，Java 的 final methods（方法）和 C# 的 sealed），现在你有了一个在 C++ 中的到类似行为的想法。

第二，你可能需要最小化 Widget 的 compilation dependencies（编译依赖）。如果 Widget 从 Timer 继承，在 Widget 被编译的时候 Timer 的 definition（定义）必须是可用的，所以定义 Widget 的文件可能不得不 #include Timer.h。另一方面，如果 WidgetTimer 移出 Widget 而 Widget 只包含一个指向一个 WidgetTimer 的 pointer（指针），Widget 就可以只需要 WidgetTimer class（类）的一个简单的 declaration（声明）；为了使用 Timer 它不需要 #include 任何东西。对于大型系统，这样的隔离可能非常重要（关于 minimizing compilation dependencies（最小化编译依赖）的细节，参见 Item 31）。

我早些时候谈及 private inheritance（私有继承）主要用武之地是当一个将要成为 derived class（派生类）的类需要访问将要成为 base class（基类）的类的 protected parts（保护构件），或者希望重定义一个或多个它的 virtual functions（虚拟函数），但是 classes（类）之间的概念上的关系却是 is-implemented-in-terms-of，而不是 is-a。然而，我也说过有一种涉及 space optimization（空间最优化）的极端情况可能会使你倾向于 private inheritance（私有继承），而不是 composition（复合）。

这个极端情况确实非常尖锐：它仅仅适用于你处理一个其中没有数据的 class（类）的时候。这样的 classes（类）没有 non-static data members（非静态数据成员）；没有 virtual functions（虚函数）（因为存在这样的函数会在每一个 object（对象）中增加一个 vptr —— 参见 Item 7）；也没有 virtual base classes（虚拟基类）（因为这样的 base classes（基类）也会引起 size overhead（大小成本）——参见 Item 40）。在理论上，这样的 empty classes（空类）的 objects（对象）应该不占用空间，因为没有 per-object（逐对象）的数据需要存储。然而，由于 C++ 天生的技术上的原因，freestanding objects（独立对象）必须有 non-zero size（非零大小），所以如果你这样做，

```
class Empty {};                      // has no data, so objects should
                                     // use no memory
class HoldsAnInt {                   // should need only space for an int
private:
  int x;
  Empty e;                           // should require no memory
};
```

你将发现 sizeof(HoldsAnInt) > sizeof(int)；一个 Empty data member（空数据成员）需要存储。对以大多数编译器，sizeof(Empty) 是 1，这是因为 C++ 法则反对 zero-size 的 freestanding objects（独立对象）一般是通过在 "empty" objects（“空”对象）中插入一个 char 完成的。然而，alignment requirements（对齐需求）（参见 Item 50）可能促使编译器向类似 HoldsAnInt 的 classes（类）中增加填充物，所以，很可能 HoldsAnInt objects 得到的不仅仅是一个 char 的大小，实际上它们可能会扩张到足以占据第二个 int 的位置。（在我测试过的所有编译器上，这毫无例外地发生了。）

但是也许你已经注意到我小心翼翼地说 "freestanding" objects（“独立”对象）必然不会有 zero size。这个约束不适用于 base class parts of derived class objects（派生类对象的基类构件），因为它们不是独立的。如果你用从 Empty 继承代替包含一个此类型的 object（对象），

```
class HoldsAnInt: private Empty {
private:
  int x;
};
```

你几乎总是会发现 sizeof(HoldsAnInt) == sizeof(int)。这个东西以 empty base optimization (EBO)（空基优化）闻名，而且它已经被我测试过的所有编译器实现。如果你是一个空间敏感的客户的库开发者，EBO 就值得了解。同样值得了解的是 EBO 通常只在 single inheritance（单继承）下才可行。支配 C++ object layout（C++ 对象布局）的规则通常意味着 EBO 不适用于拥有多于一个 base（基）的 derived classes（派生类）。

在实践中，"empty" classes（“空”类）并不真的为空。虽然他们绝对不会有 non-static data members（非静态数据成员），但它们经常会包含 typedefs，enums（枚举），static data members（静态数据成员），或 non-virtual functions（非虚拟函数）。STL 有很多包含有用的 members（成员）（通常是 typedefs）的专门的 empty classes（空类），包括 base classes（基类）unary_function 和 binary_function，user-defined function objects（用户定义函数对象）通常从这些 classes（类）继承而来。感谢 EBO 的普遍实现，这样的继承很少增加 inheriting classes（继承来的类）的大小。

尽管如此，我们还是要回归基础。大多数 classes（类）不是空的，所以 EBO 很少会成为 private inheritance（私有继承）的一个合理的理由。此外，大多数 inheritance（继承）相当于 is-a，而这正是 public inheritance（公有继承）而非 private（私有）所做的事。composition（复合）和 private inheritance（私有继承）两者都意味着 is-implemented-in-terms-of（是根据……实现的），但是 composition（复合）更易于理解，所以你应该尽你所能使用它。

private inheritance（私有继承）更可能在以下情况中成为一种设计策略，当你要处理的两个 classes（类）不具有 is-a（是一个）的关系，而且其中的一个还需要访问另一个的 protected members（保护成员）或需要重定义一个或更多个它的 virtual functions（虚拟函数）。甚至在这种情况下，我们也看到 public inheritance 和 containment 的混合使用通常也能产生你想

要的行为，虽然有更大的设计复杂度。谨慎使用 private inheritance（私有继承）意味着在使用它的时候，已经考虑过所有的可选方案，只有它才是你的软件中明确表示两个 classes（类）之间关系的最佳方法。

Things to Remember

- private inheritance（私有继承）意味着 is-implemented-in-terms of（是根据……实现的）。它通常比 composition（复合）更低级，但当一个 derived class（派生类）需要访问 protected base class members（保护基类成员）或需要重定义 inherited virtual functions（继承来的虚拟函数）时它就是合理的。
- 与 composition（复合）不同，private inheritance（私有继承）能使 empty base optimization（空基优化）有效。这对于致力于最小化 object sizes（对象大小）的库开发者来说可能是很重要的。

Item 40: 谨慎使用 multiple inheritance（多继承）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

触及 multiple inheritance (MI)（多继承）的时候，C++ 社区就会鲜明地分裂为两个基本的阵营。一个阵营认为如果 single inheritance (SI)（单继承）是有好处的，multiple inheritance（多继承）一定更有好处。另一个阵营认为 single inheritance（单继承）有好处，但是多继承引起的麻烦使它得不偿失。在这个 Item 中，我们的主要目的是理解在 MI 问题上的这两种看法。

首要的事情之一是要承认当将 MI 引入设计领域时，就有可能从多于一个的 base class（基类）中继承相同的名字（例如，函数，typedef，等等）。这就为歧义性提供了新的时机。例如：

```
class BorrowableItem {             // something a library lets you borrow
public:
  void checkOut();                 // check the item out from the library

  ...
};

class ElectronicGadget {
private:
  bool checkOut() const;           // perform self-test, return whether

  ...                              // test succeeds
};

class MP3Player:                   // note MI here
  public BorrowableItem,           // (some libraries loan MP3 players)
  public ElectronicGadget
{ ... };                           // class definition is unimportant

MP3Player mp;

mp.checkOut();                     // ambiguous! which checkOut?
```

注意这个例子，即使两个函数中只有一个是可访问的，对 checkOut 的调用也是有歧义的。（checkOut 在 BorrowableItem 中是 public（公有）的，但在 ElectronicGadget 中是 private（私有）的。）这与 C++ 解析 overloaded functions（重载函数）调用的规则是一致的：在看到一个函数的是否可访问之前，C++ 首先确定与调用匹配最好的那个函数。只有在确定了 best-match function（最佳匹配函数）之后，才检查可访问性。这目前的情况下，两个 checkOuts 具有相同的匹配程度，所以就不存在最佳匹配。因此永远也不会检查到 ElectronicGadget::checkOut 的可访问性。

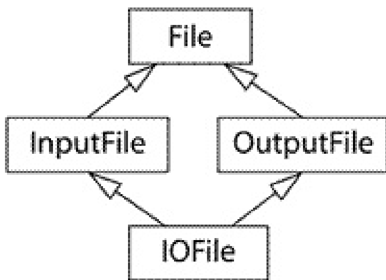
为了消除歧义性，你必须指定哪一个 base class（基类）的函数被调用：

```
mp.BorrowableItem::checkOut(); // ah, that checkOut...
```

当然，你也可以尝试显式调用 ElectronicGadget::checkOut，但这样做会有一个 "you're trying to call a private member function"（你试图调用一个私有成员函数）错误代替歧义性错误。

multiple inheritance（多继承）仅仅意味着从多于一个的 base class（基类）继承，但是在还有 higher-level base classes（更高层次基类）的 hierarchies（继承体系）中出现 MI 也并不罕见。这会导致有时被称为 "deadly MI diamond"（致命的多继承菱形）的后果。

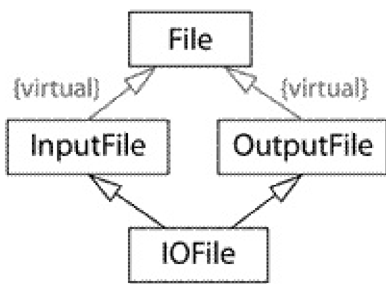
```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



你拥有一个“在一个 base class（基类）和一个 derived class（派生类）之间有多于一条路径的 inheritance hierarchy（继承体系）”（就像上面在 File 和 IOFile 之间，有通过 InputFile 和 OutputFile 的两条路径）的任何时候，你都必须面对是否需要为每一条路径复制 base class（基类）中的 data members（数据成员）的问题。例如，假设 File class 有一个 data members（数据成员）fileName。IOFile 中应该有这个 field（字段）的多少个拷贝呢？一方面，它从它的每一个 base classes（基类）继承一个拷贝，这就暗示 IOFile 应该有两个 fileName data members（数据成员）。另一方面，简单的逻辑告诉我们一个 IOFile object（对象）应该仅有一个 file name（文件名），所以通过它的两个 base classes（基类）继承来的 fileName field（字段）不应该被复制。

C++ 在这个争议上没有自己的立场。它恰当地支持两种选项，虽然它的缺省方式是执行复制。如果那不是你想要的，你必须让这个 class（类）带有一个 virtual base class（虚拟基类）的数据（也就是 File）。为了做到这一点，你要让从它直接继承的所有的 classes（类）使用 virtual inheritance（虚拟继承）：

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



标准 C++ 库包含一个和此类似的 MI hierarchy（继承体系），只是那个 classes（类）是 class templates（类模板），名字是 basic_ios，basic_istream，basic_ostream 和 basic_iostream，而不是 File，InputFile，OutputFile 和 IOFile。

从正确行为的观点看，public inheritance（公有继承）应该总是 virtual（虚拟）的。如果这是唯一的观点，规则就变得简单了：你使用 public inheritance（公有继承）的任何时候，都使用 virtual public inheritance（虚拟公有继承）。唉，正确性不是唯一的视角。避免 inherited fields（继承来的字段）复制需要在编译器的一部分做一些 behind-the-scenes legerdemain（幕后的戏法），而结果是从使用 virtual inheritance（虚拟继承）的 classes（类）创建的 objects（对象）通常比不使用 virtual inheritance（虚拟继承）的要大。访问 virtual base classes（虚拟基类）中的 data members（数据成员）也比那些 non-virtual base classes（非虚拟基类）中的要慢。编译器与编译器之间有一些细节不同，但基本的要点很清楚：virtual inheritance costs（虚拟继承要付出成本）。

它也有一些其它方面的成本。支配 initialization of virtual base classes（虚拟基类初始化）的规则比 non-virtual bases（非虚拟基类）的更加复杂而且更不直观。初始化一个 virtual base（虚拟基）的职责由 hierarchy（继承体系）中 most derived class（层次最低的派生类）承担。这个规则中包括的含义：（1） 从需要 initialization（初始化）的 virtual bases（虚拟基）派生的 classes（类）必须知道它们的 virtual bases（虚拟基），无论它距离那个 bases（基）有多远；（2） 当一个新的 derived class（派生类）被加入继承体系时，它必须为它的 virtual bases（虚拟基）（包括直接的和间接的）承担 initialization responsibilities（初始化职责）。

我对于 virtual base classes（虚拟基类）（也就是 virtual inheritance（虚拟继承））的建议很简单。首先，除非必需，否则不要使用 virtual bases（虚拟基）。缺省情况下，使用 non-virtual inheritance（非虚拟继承）。第二，如果你必须使用 virtual base classes（虚拟基类），试着避免在其中放置数据。这样你就不必在意它的 initialization（初始化）（以及它的 turns out（清空），assignment（赋值））规则中的一些怪癖。值得一提的是 Java 和 .NET 中的 Interfaces（接口）不允许包含任何数据，它们在很多方面可以和 C++ 中的 virtual base classes（虚拟基类）相比照。

现在我们使用下面的 C++ Interface class（接口类）（参见 Item 31）来为 persons（人）建模：

```
class IPerson {
public:
  virtual ~IPerson();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};
```

IPerson 的客户只能使用 IPerson 的 pointers（指针）和 references（引用）进行编程，因为 abstract classes（抽象类）不能被实例化。为了创建能被当作 IPerson objects（对象）使用的 objects（对象），IPerson 的客户使用 factory functions（工厂函数）（再次参见 Item 31）instantiate（实例化）从 IPerson 派生的 concrete classes（具体类）：

```
// factory function to create a Person object from a unique database ID;
// see Item 18 for why the return type isn't a raw pointer
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

// function to get a database ID from the user
DatabaseID askUserForDatabaseID();


DatabaseID id(askUserForDatabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id));    // create an object
                                                     // supporting the
                                                     // IPerson interface

...                                                  // manipulate *pp via
                                                     // IPerson's member
                                                     // functions
```

但是 makePerson 怎样创建它返回的 pointers（指针）所指向的 objects（对象）呢？显然，必须有一些 makePerson 可以实例化的从 IPerson 派生的 concrete class（具体类）。

假设这个 class（类）叫做 CPerson。作为一个 concrete class（具体类），CPerson 必须提供它从 IPerson 继承来的 pure virtual functions（纯虚拟函数）的 implementations（实现）。它可以从头开始写，但利用包含大多数或全部必需品的现有组件更好一些。例如，假设一个老式的 database-specific class（老式的数据库专用类）PersonInfo 提供了 CPerson 所需要的基本要素：

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  ...

private:
  virtual const char * valueDelimOpen() const;      // see
  virtual const char * valueDelimClose() const;     // below
  ...
};
```

你可以看出这是一个老式的 class（类），因为 member functions（成员函数）返回 const char*s 而不是 string objects（对象）。尽管如此，如果鞋子合适，为什么不穿呢？这个 class（类）的 member functions（成员函数）的名字暗示结果很可能会非常合适。

你突然发现 PersonInfo 是设计用来帮助以不同的格式打印 database fields（数据库字段）的，每一个字段的值的开始和结尾通过指定的字符串定界。缺省情况下，字段值开始和结尾定界符是方括号，所以字段值 "Ring-tailed Lemur" 很可能被安排成这种格式：

```
[Ring-tailed Lemur]
```

根据方括号并非满足 PersonInfo 的全体客户的期望的事实，virtual functions（虚拟函数）valueDelimOpen 和 valueDelimClose 允许 derived classes（派生类）指定它们自己的开始和结尾定界字符串。PersonInfo 的 member functions（成员函数）的 implementations（实现）调用这些 virtual functions（虚拟函数）在它们返回的值上加上适当的定界符。作为一个例子使用 PersonInfo::theName，代码如下：

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                       // default opening delimiter
}

const char * PersonInfo::valueDelimClose() const
{
  return "]";                       // default closing delimiter
}

const char * PersonInfo::theName() const
{
  // reserve buffer for return value; because this is
  // static, it's automatically initialized to all zeros
  static char value[Max_Formatted_Field_Value_Length];

  // write opening delimiter
  std::strcpy(value, valueDelimOpen());

  append to the string in value this object's   name field (being careful
  to avoid buffer overruns!)

  // write closing delimiter
  std::strcat(value, valueDelimClose());

  return value;
}
```

有人可能会质疑 PersonInfo::theName 的陈旧的设计（特别是一个 fixed-size static buffer（固定大小静态缓冲区）的使用，这样的东西发生 overrun（越界）和 threading（线程）问题是比较普遍的——参见 Item 21），但是请把这样的问题放到一边而注意这里：theName 调用 valueDelimOpen 生成它要返回的 string（字符串）的开始定界符，然后它生成名字值本身，然后它调用 valueDelimClose。

因为 valueDelimOpen 和 valueDelimClose 是 virtual functions（虚拟函数），theName 返回的结果不仅依赖于 PersonInfo，也依赖于从 PersonInfo 派生的 classes（类）。

对于 CPerson 的实现者，这是好消息，因为当细读 IPerson documentation（文档）中的 fine print（晦涩的条文）时，你发现 name 和 birthDate 需要返回未经修饰的值，也就是，不允许有定界符。换句话说，如果一个人的名字叫 Homer，对那个人的 name 函数的一次调用应该返回 "Homer"，而不是 "[Homer]"。

CPerson 和 PersonInfo 之间的关系是 PersonInfo 碰巧有一些函数使得 CPerson 更容易实现。这就是全部。因而它们的关系就是 is-implemented-in-terms-of，而我们知道有两种方法可以表现这一点：经由 composition（复合）（参见 Item 38）和经由 private inheritance（私有继承）（参见 Item 39）。Item 39 指出 composition（复合）是通常的首选方法，但如果 virtual functions（虚拟函数）要被重定义，inheritance（继承）就是必不可少的。在当前情况下，CPerson 需要重定义 valueDelimOpen 和 valueDelimClose，所以简单的 composition（复合）做不到。最直截了当的解决方案是让 CPerson 从 PersonInfo privately inherit（私有继承），虽然 Item 39 说过只要多做一点工作，则 CPerson 也能用 composition（复合）和 inheritance（继承）的组合有效地重定义 PersonInfo 的 virtuals（虚拟函数）。这里，我们用 private inheritance（私有继承）。

但是 CPerson 还必须实现 IPerson interface（接口），而这被称为 public inheritance（公有继承）。这就引出一个 multiple inheritance（多继承）的合理应用：组合 public inheritance of an interface（一个接口的公有继承）和 private inheritance of an implementation（一个实现的私有继承）：

```
class IPerson {                           // this class specifies the
public:                                   // interface to be implemented
  virtual ~IPerson();

  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                 // used below; details are
                                          // unimportant

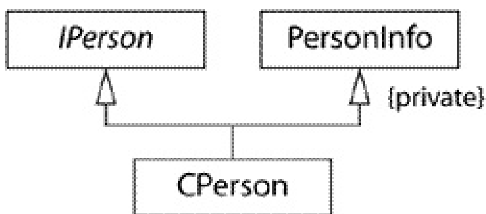
class PersonInfo {                        // this class has functions
public:                                   // useful in implementing
  explicit PersonInfo(DatabaseID pid);    // the IPerson interface
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;

  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;
  ...
};

class CPerson: public IPerson, private PersonInfo {     // note use of MI
public:
  explicit CPerson(    DatabaseID pid): PersonInfo(pid) {}
  virtual std::string name() const                      // implementations
  { return PersonInfo::theName(); }                     // of the required
                                                        // IPerson member
  virtual std::string birthDate() const                 // functions
  { return PersonInfo::theBirthDate(); }
private:                                                // redefinitions of
  const char * valueDelimOpen() const { return ""; }    // inherited virtual
  const char * valueDelimClose() const { return ""; }   // delimiter
};
```

在 UML 中，这个设计看起来像这样：



这个例子证明 MI 既是有用的，也是可理解的。

时至今日，multiple inheritance（多继承）不过是 object-oriented toolbox（面向对象工具箱）里的又一种工具而已，典型情况下，它的使用和理解更加复杂，所以如果你得到一个或多或少等同于一个 MI 设计的 SI 设计，则 SI 设计总是更加可取。如果你能拿出来的仅有的设计包含 MI，你应该更加用心地考虑一下——总会有一些方法使得 SI 也能做到。但同时，MI 有时是最清晰的，最易于维护的，最合理的完成工作的方法。在这种情况下，毫不畏惧地使用它。只是要确保谨慎地使用它。

Things to Remember

- multiple inheritance（多继承）比 single inheritance（单继承）更复杂。它能导致新的歧义问题和对 virtual inheritance（虚拟继承）的需要。

- virtual inheritance（虚拟继承）增加了 size（大小）和 speed（速度）成本，以及 initialization（初始化）和 assignment（赋值）的复杂度。当 virtual base classes（虚拟基类）没有数据时它是最适用的。
- multiple inheritance（多继承）有合理的用途。一种方案涉及组合从一个 Interface class（接口类）的 public inheritance（公有继承）和从一个有助于实现的 class（类）的 private inheritance（私有继承）。

Item 41: 理解 implicit interfaces（隐式接口）和 compile-time polymorphism（编译期多态）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

object-oriented programming（面向对象编程）的世界是围绕着 explicit interfaces（显式接口）和 runtime polymorphism（执行期多态）为中心的。例如，给出下面这个（没有什么意义的）class（类），

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);                 // see Item 25

  ...
};
```

以及这个（同样没有什么意义的）function（函数），

```
void doProcessing(Widget& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
      Widget temp(w);
      temp.normalize();
      temp.swap(w);
  }
}
```

我们可以这样谈论 doProcessing 中的 w：

- 因为 w 被声明为 Widget 类型的引用，w 必须支持 Widget interface（接口）。我们可以在源代码中找到这个 interface（接口）（例如，Widget 的 .h 文件）以看清楚它是什么样子的，所以我们称其为一个 explicit interface（显式接口）——它在源代码中显式可见。
- 因为 Widget 的一些 member functions（成员函数）是虚拟的，w 对这些函数的调用就表现为 runtime polymorphism（执行期多态）：被调用的特定函数在执行期基于 w 的 dynamic type（动态类型）来确定（参见 Item 37）。

templates（模板）和 generic programming（泛型编程）的世界是根本不同的。在那个世界，explicit interfaces（显式接口）和 runtime polymorphism（执行期多态）继续存在，但是它们不那么重要了。作为替代，把 implicit interfaces（隐式接口）和 compile-time

polymorphism（编译期多态）推到了前面。为了了解这是怎样一种情况，看一下当我们把 doProcessing 从一个 function（函数）转为一个 function template（函数模板）时会发生什么：

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
     T temp(w);
     temp.normalize();
     temp.swap(w);
  }
}
```

现在我们可以如何谈论 doProcessing 中的 w 呢？

- w 必须支持的 interface（接口）是通过 template（模板）中在 w 身上所执行的操作确定的。在本例中，它显现为 w 的 type (T) 必须支持 size，normalize 和 swap member functions（成员函数）；copy construction（拷贝构造函数）（用于创建 temp）；以及对不等于的比较（用于和 someNastyWidget 之间的比较）。我们将在以后看到这并不很精确，但是对于现在来说它已经足够正确了。重要的是这一系列必须有效地适合于模板编译的表达式是 T 必须支持的 implicit interface（隐式接口）。

- 对诸如 operator> 和 operator!= 这样的包含 w 的函数的调用可能伴随 instantiating templates（实例化模板）以使这些调用成功。这样的 instantiation（实例化）发生在编译期间。因为用不同的 template parameters（模板参数）实例化 function templates（函数模板）导致不同的函数被调用，因此以 compile-time polymorphism（编译期多态）著称。

即使你从没有使用过模板，你也应该熟悉 runtime（运行期）和 compile-time polymorphism（编译期多态）之间的区别，因为它类似于确定一系列重载函数中哪一个应该被调用的过程（这个发生在编译期）和 virtual function（虚拟函数）调用的 dynamic binding（动态绑定）（这个发生在运行期）之间的区别。explicit（显式）和 implicit interfaces（隐式接口）之间的区别是与 template（模板）有关的新内容，需要对他进行近距离的考察。

一个 explicit interface（显式接口）由 function signatures（函数识别特征）组成，也就是说，函数名，参数类型，返回值，等等。例如，Widget class（类）的 public interface（显式接口），

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);
};
```

由一个 constructor（构造函数），一个 destructor（析构函数），以及函数 size，normalize 和 swap 组成，再加上 parameter types（参数类型），return types（返回类型）和这些函数的 constnesses（常量性）。（它也包括 compiler-generated（编译器生成）的 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）——参见 Item 5。）它还可能包含 typedefs，还有，如果你胆大包天敢于违背 Item 22 的让 data members（数据成员）private（私有）的建议，即使在这种情况下，这些 data members（数据成员）也不是。

一个 implicit interface（隐式接口）有很大不同。它不是基于 function signatures（函数识别特征）的。它是由 valid expressions（合法表达式）组成的。再看一下在 doProcessing template 开始处的条件：

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
  ...
```

对于 T（w 的类型）的 implicit interface（隐式接口）看起来有如下这些约束：

- 它必须提供一个名为 size 的返回一个正数值的 member function（成员函数）。
- 它必须支持一个用于比较两个类型 T 的对象的 operator!= 函数。（这里，我们假定 someNastyWidget 的类型为 T。）

由于 operator overloading（运算符重载）的可能性，这两个约束都不必满足。是的，T 必须支持一个 size member function（成员函数），值得提及的是虽然这个函数可以是从一个 base class（基类）继承来的。但是这个 member function（成员函数）不需要返回一个整数类型。它甚至不需要返回一个数值类型。对于这种情况，它甚至不需要返回一个定义了 operator> 的类型！它要做的全部就是返回类型 X 的一个 object（对象），有一个 operator> 可以用一个类型为 X 的 object（对象）和一个 int（因为 10 为 int 类型）来调用。这个 operator> 不需要取得一个类型 X 的参数，因为它可以取得一个类型 Y 的参数，只要在类型 X 的 objects（对象）和类型 Y 的 objects（对象）之间有一个 implicit conversion（隐式转型）就可以了！

类似地，T 支持 operator!= 也是没有必要的，因为如果 operator!= 取得一个类型 X 的 objects（对象）和一个类型 Y 的 objects（对象）是可接受的一样。只要 T 能转型为 X，而 someNastyWidget 的类型能够转型为 Y，对 operator!= 的调用就是合法的。

（一个旁注：此处的分析没有考虑 operator&& 被重载的可能性，这会将上面的表达式的含义从与转换到某些大概完全不同的东西。）

第一次考虑 implicit interfaces（隐式接口）的时候，大多数人都会头疼，但是他们真的不需要阿司匹林。implicit interfaces（隐式接口）简单地由一套 valid expressions（合法表达式）构成。这些表达式自身看起来可能很复杂，但是它们施加的约束通常是简单易懂的。例如，

给出这个条件，

```
if (w.size() > 10 && w != someNastyWidget) ...
```

关于 functions size，operator>，operator&& 或 operator!= 上的约束很难说出更多的东西，但是要识别出整个表达式的约束是非常简单的。一个 if 语句的条件部分必须是一个 boolean expression（布尔表达式），所以不管 "w.size() > 10 && w != someNastyWidget" 所产生的类型涉及到的精确类型，它必须与 bool 兼容。这就是 template（模板）doProcessing 施加于它的 type parameter（类型参数）T 之上的 implicit interface（隐式接口）的一部分。被 doProcessing 需要的 interface（接口）的其余部分是 copy constructor（拷贝构造函数），normalize 和 swap 的调用对于类型 T 的 objects（对象）来说必须是合法的。

implicit interface（隐式接口）对 template（模板）的 parameters（参数）施加的影响正像 explicit interfaces（显式接口）对一个 class（类）的 objects（对象）施加的影响，而且这两者都在编译期间被检查。正像你不能用与它的 class（类）提供的 explicit interface（显式接口）矛盾的方法使用 object（对象）（代码无法编译）一样，除非一个 object（对象）支持 template（模板）所需要的 implicit interface（隐式接口），否则你就不能在一个 template（模板）中试图使用这个 object（对象）（代码还是无法编译）。

Things to Remember

- classes（类）和 templates（模板）都支持 interfaces（接口）和 polymorphism（多态）。
- 对于 classes（类），interfaces（接口）是 explicit（显式）的并以 function signatures（函数识别特征）为中心的。polymorphism（多态性）通过 virtual functions（虚拟函数）出现在运行期。
- 对于 template parameters（模板参数），interfaces（接口）是 implicit（隐式）的并基于 valid expressions（合法表达式）。polymorphism（多态性）通过 template instantiation（模板实例化）和 function overloading resolution（函数重载解析）出现在编译期。

Item 42: 理解 typename 的两个含义

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

问题：在下面的 template declarations（模板声明）中 class 和 typename 有什么不同？

```
template<class T> class Widget;                 // uses "class"

template<typename T> class Widget;              // uses "typename"
```

答案：没什么不同。在声明一个 template type parameter（模板类型参数）的时候，class 和 typename 意味着完全相同的东西。一些程序员更喜欢在所有的时间都用 class，因为它更容易输入。其他人（包括我本人）更喜欢 typename，因为它暗示着这个参数不必要是一个 class type（类类型）。少数开发者在任何类型都被允许的时候使用 typename，而把 class 保留给仅接受 user-defined types（用户定义类型）的场合。但是从 C++ 的观点看，class 和 typename 在声明一个 template parameter（模板参数）时意味着完全相同的东西。

然而，C++ 并不总是把 class 和 typename 视为等同的东西。有时你必须使用 typename。为了理解这一点，我们不得不讨论你会在一个 template（模板）中涉及到的两种名字。

假设我们有一个函数的模板，它能取得一个 STL-compatible container（STL 兼容容器）中持有的能赋值给 ints 的对象。进一步假设这个函数只是简单地打印它的第二个元素的值。它是一个用糊涂的方法实现的糊涂的函数，而且就像我下面写的，它甚至不能编译，但是请将这些事先放在一边——有一种方法能发现我的愚蠢：

```
template<typename C>                               // print 2nd element in
void print2nd(const C& container)                  // container;
{                                                  // this is not valid C++!
  if (container.size() >= 2) {
     C::const_iterator iter(container.begin());   // get iterator to 1st element
     ++iter;                                       // move iter to 2nd element
     int value = *iter;                            // copy that element to an int
     std::cout << value;                           // print the int
  }
}
```

我突出了这个函数中的两个 local variables（局部变量），iter 和 value。iter 的类型是 C::const_iterator，一个依赖于 template parameter（模板参数）C 的类型。一个 template（模板）中的依赖于一个 template parameter（模板参数）的名字被称为 dependent names（依赖名字）。当一个 dependent names（依赖名字）嵌套在一个 class（类）的内部时，我称它为 nested dependent name（嵌套依赖名字）。C::const_iterator 是一个 nested

dependent name（嵌套依赖名字）。实际上，它是一个 nested dependent type name（嵌套依赖类型名），也就是说，一个涉及到一个 type（类型）的 nested dependent name（嵌套依赖名字）。

print2nd 中的另一个 local variable（局部变量）value 具有 int 类型。int 是一个不依赖于任何 template parameter（模板参数）的名字。这样的名字以 non-dependent names（非依赖名字）闻名。（我想不通为什么他们不称它为 independent names（无依赖名字）。如果，像我一样，你发现术语 "non-dependent" 是一个令人厌恶的东西，你就和我产生了共鸣，但是 "non-dependent" 就是这类名字的术语，所以，像我一样，转转眼睛放弃你的自我主张。）

nested dependent name（嵌套依赖名字）会导致解析困难。例如，假设我们更加愚蠢地以这种方法开始 print2nd：

```
template<typename C>
void print2nd(const C& container)
{
  C::const_iterator * x;
  ...
}
```

这看上去好像是我们将 x 声明为一个指向 C::const_iterator 的 local variable（局部变量）。但是它看上去如此仅仅是因为我们知道 C::const_iterator 是一个 type（类型）。但是如果 C::const_iterator 不是一个 type（类型）呢？如果 C 有一个 static data member（静态数据成员）碰巧就叫做 const_iterator 呢？再如果 x 碰巧是一个 global variable（全局变量）的名字呢？在这种情况下，上面的代码就不是声明一个 local variable（局部变量），而是成为 C::const_iterator 乘以 x！当然，这听起来有些愚蠢，但它是可能的，而编写 C++ 解析器的人必须考虑所有可能的输入，甚至是愚蠢的。

直到 C 成为已知之前，没有任何办法知道 C::const_iterator 到底是不是一个 type（类型），而当 template（模板）print2nd 被解析的时候，C 还不是已知的。C++ 有一条规则解决这个歧义：如果解析器在一个 template（模板）中遇到一个 nested dependent name（嵌套依赖名字），它假定那个名字不是一个 type（类型），除非你用其它方式告诉它。缺省情况下，nested dependent name（嵌套依赖名字）不是 types（类型）。（对于这条规则有一个例外，我待会儿告诉你。）

记住这个，再看看 print2nd 的开头：

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size() >= 2) {
     C::const_iterator iter(container.begin());   // this name is assumed to
     ...                                          // not be a type
```

这为什么不是合法的 C++ 现在应该很清楚了。iter 的 declaration（声明）仅仅在 C::const_iterator 是一个 type（类型）时才有意义，但是我们没有告诉 C++ 它是，而 C++ 就假定它不是。要想转变这个形势，我们必须告诉 C++ C::const_iterator 是一个 type（类

型）。我们将 typename 放在紧挨着它的前面来做到这一点：

```
template<typename C>                           // this is valid C++
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());
    ...
  }
}
```

通用的规则很简单：在你涉及到一个在 template（模板）中的 nested dependent type name（嵌套依赖类型名）的任何时候，你必须把单词 typename 放在紧挨着它的前面。（重申一下，我待会儿要描述一个例外。）

typename 应该仅仅被用于标识 nested dependent type name（嵌套依赖类型名）；其它名字不应该用它。例如，这是一个取得一个 container（容器）和这个 container（容器）中的一个 iterator（迭代器）的 function template（函数模板）：

```
template<typename C>                   // typename allowed (as is "class")
void f(const C& container,             // typename not allowed
     typename C::iterator iter);       // typename required
```

C 不是一个 nested dependent type name（嵌套依赖类型名）（它不是嵌套在依赖于一个 template parameter（模板参数）的什么东西内部的），所以在声明 container 时它不必被 typename 前置，但是 C::iterator 是一个 nested dependent type name（嵌套依赖类型名），所以它必需被 typename 前置。

"typename must precede nested dependent type names"（“typename 必须前置于嵌套依赖类型名”）规则的例外是 typename 不必前置于在一个 list of base classes（基类列表）中的或者在一个 member initialization list（成员初始化列表）中作为一个 base classes identifier（基类标识符）的 nested dependent type name（嵌套依赖类型名）。例如：

```
template<typename T>
class Derived: public Base<T>::Nested { // base class list: typename not
public:                                 // allowed
  explicit Derived(int x)
  : Base<T>::Nested(x)                  // base class identifier in mem
  {                                     // init. list: typename not allowed

    typename Base<T>::Nested temp;      // use of nested dependent type
    ...                                 // name not in a base class list or
  }                                     // as a base class identifier in a
  ...                                   // mem. init. list: typename required
};
```

这样的矛盾很令人讨厌，但是一旦你在经历中获得一点经验，你几乎不会在意它。

让我们来看最后一个 typename 的例子，因为它在你看到的真实代码中具有代表性。假设我们在写一个取得一个 iterator（迭代器）的 function template（函数模板），而且我们要做一个 iterator（迭代器）指向的 object（对象）的局部拷贝 temp，我们可以这样做：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typename std::iterator_traits<IterT>::value_type temp(*iter);
  ...
}
```

不要让 std::iterator_traits<IterT>::value_type 吓倒你。那仅仅是一个 standard traits class（标准特性类）（参见 Item 47）的使用，用 C++ 的说法就是 "the type of thing pointed to by objects of type IterT"（“被类型为 IterT 的对象所指向的东西的类型”）。这个语句声明了一个与 IterT objects 所指向的东西类型相同的 local variable（局部变量）(temp)，而且用 iter 所指向的 object（对象）对 temp 进行了初始化。如果 IterT 是 vector<int>::iterator，temp 就是 int 类型。如果 IterT 是 list<string>::iterator，temp 就是 string 类型。因为 std::iterator_traits<IterT>::value_type 是一个 nested dependent type name（嵌套依赖类型名）（value_type 嵌套在 iterator_traits<IterT> 内部，而且 IterT 是一个 template parameter（模板参数）），我们必须让它被 typename 前置。

如果你觉得读 std::iterator_traits<IterT>::value_type 令人讨厌，就想象那个与它相同的东西来代表它。如果你像大多数程序员，对多次输入它感到恐惧，那么你就需要创建一个 typedef。对于像 value_type 这样的 traits member names（特性成员名）（再次参见 Item 47 关于 traits 的资料），一个通用的惯例是 typedef name 与 traits member name 相同，所以这样的一个 local typedef 通常定义成这样：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
  typedef typename std::iterator_traits<IterT>::value_type value_type;

  value_type temp(*iter);
  ...
}
```

很多程序员最初发现 "typedef typename" 并列不太和谐，但它是涉及 nested dependent type names（嵌套依赖类型名）规则的一个合理的附带结果。你会相当快地习惯它。你毕竟有着强大的动机。你输入 typename std::iterator_traits<IterT>::value_type 需要多少时间？

作为结束语，我应该提及编译器与编译器之间对围绕 typename 的规则的执行情况的不同。一些编译器接受必需 typename 时它却缺失的代码；一些编译器接受不许 typename 时它却存在的代码；还有少数的（通常是老旧的）会拒绝 typename 出现在它必需出现的地方。这就意味着 typename 和 nested dependent type names（嵌套依赖类型名）的交互作用会导致一些轻微的可移植性问题。

Things to Remember

- 在声明 template parameters（模板参数）时，class 和 typename 是可互换的。

* 用 typename 去标识 nested dependent type names（嵌套依赖类型名），在 base class lists（基类列表）中或在一个 member initialization list（成员初始化列表）中作为一个 base class identifier（基类标识符）时除外。

Item 43: 了解如何访问 templatized base classes（模板化基类）中的名字

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们要写一个应用程序，它可以把消息传送到几个不同的公司去。消息既可以以加密方式也可以以明文（不加密）的方式传送。如果我们有足够的信息在编译期间确定哪个消息将要发送给哪个公司，我们就可以用一个 template-based（模板基）来解决问题：

```
class CompanyA {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};

class CompanyB {
public:
  ...
  void sendCleartext(const std::string& msg);
  void sendEncrypted(const std::string& msg);
  ...
};
...                                       // classes for other companies

class MsgInfo { ... };                    // class for holding information
                                          // used to create a message
template<typename Company>
class MsgSender {
public:
  ...                                     // ctors, dtor, etc.

  void sendClear(const MsgInfo& info)
  {
    std::string msg;
    create msg from info;

    Company c;
    c.sendCleartext(msg);
  }

  void sendSecret(const MsgInfo& info)   // similar to sendClear, except
  { ... }                                 // calls c.sendEncrypted
};
```

这个能够很好地工作，但是假设我们有时需要在每次发送消息的时候把一些信息记录到日志中。通过一个 derived class（派生类）可以很简单地增加这个功能，下面这个似乎是一个合理的方法：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...                                   // ctors, dtor, etc.
  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    sendClear(info);                    // call base class function;
                                        // this code will not compile!
    write "after sending" info to the log;
  }
  ...

};
```

注意 derived class（派生类）中的 message-sending function（消息发送函数）的名字 (sendClearMsg) 与它的 base class（基类）中的那个（在那里，它被称为 sendClear）不同。这是一个好的设计，因为它避开了 hiding inherited names（隐藏继承来的名字）的问题（参见 Item 33）和重定义一个 inherited non-virtual function（继承来的非虚拟函数）的与生俱来的问题（参见 Item 36）。但是上面的代码不能通过编译，至少在符合标准的编译器上不能。这样的编译器会抱怨 sendClear 不存在。我们可以看见 sendClear 就在 base class（基类）中，但编译器不会到那里去寻找它。我们有必要理解这是为什么。

问题在于当编译器遇到 class template（类模板）LoggingMsgSender 的 definition（定义）时，它们不知道它从哪个 class（类）继承。当然，它是 MsgSender<Company>，但是 Company 是一个 template parameter（模板参数），这个直到更迟一些才能被确定（当 LoggingMsgSender 被实例化的时候）。不知道 Company 是什么，就没有办法知道 class（类）MsgSender<Company> 是什么样子的。特别是，没有办法知道它是否有一个 sendClear function（函数）。

为了使问题具体化，假设我们有一个要求加密通讯的 class（类）CompanyZ：

```
class CompanyZ {                             // this class offers no
public:                                      // sendCleartext function
  ...
  void sendEncrypted(const std::string& msg);
  ...
};
```

一般的 MsgSender template（模板）不适用于 CompanyZ，因为那个模板提供一个 sendClear function（函数）对于 CompanyZ objects（对象）没有意义。为了纠正这个问题，我们可以创建一个 MsgSender 针对 CompanyZ 的特化版本：

```
template<>                                 // a total specialization of
class MsgSender<CompanyZ> {                // MsgSender; the same as the
public:                                    // general template, except
  ...                                      // sendCleartext is omitted
  void sendSecret(const MsgInfo& info)
  { ... }
};
```

注意这个 class definition（类定义）开始处的 "template <>" 语法。它表示这既不是一个 template（模板），也不是一个 standalone class（独立类）。正确的说法是，它是一个用于 template argument（模板参数）为 CompanyZ 时的 MsgSender template（模板）的 specialized version（特化版本）。这以 total template specialization（完全模板特化）闻名：template（模板）MsgSender 针对类型 CompanyZ 被特化，而且这个 specialization（特化）是 total（完全）的——只要 type parameter（类型参数）被定义成了 CompanyZ，就没有剩下能被改变的其它 template's parameters（模板参数）。

已知 MsgSender 针对 CompanyZ 被特化，再次考虑 derived class（派生类）LoggingMsgSender：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    sendClear(info);                          // if Company == CompanyZ,
                                              // this function doesn't exist!
    write "after sending" info to the log;
  }
  ...

};
```

就像注释中写的，当 base class（基类）是 MsgSender<CompanyZ> 时，这里的代码是无意义的，因为那个类没有提供 sendClear function（函数）。这就是为什么 C++ 拒绝这个调用：它认识到 base class templates（基类模板）可能被特化，而这个特化不一定提供和 general template（通用模板）相同的 interface（接口）。结果，它通常会拒绝在 templatized base classes（模板化基类）中寻找 inherited names（继承来的名字）。在某种意义上，当我们从 Object-oriented C++ 跨越到 Template C++（参见 Item 1）时，inheritance（继承）会停止工作。

为了重新启动它，我们必须以某种方式使 C++ 的 "don't look in templatized base classes"（不在模板基类中寻找）行为失效。有三种方法可以做到这一点。首先，你可以在被调用的 base class functions（基类函数）前面加上 "this->"：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:

  ...

  void sendClearMsg(const MsgInfo& info)
  {
    write "before sending" info to the log;

    this->sendClear(info);                // okay, assumes that
                                          // sendClear will be inherited
    write "after sending" info to the log;
  }

  ...

};
```

第二，你可以使用一个 using declaration，如果你已经读过 Item 33，这应该是你很熟悉的一种解决方案。那个 Item 解释了 using declarations 如何将被隐藏的 base class names（基类名字）引入到一个 derived class（派生类）领域中。因此我们可以这样写 sendClearMsg：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;   // tell compilers to assume
  ...                                    // that sendClear is in the
                                         // base class
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    sendClear(info);                     // okay, assumes that
    ...                                  // sendClear will be inherited
  }

  ...
};
```

（虽然 using declaration 在这里和 Item 33 中都可以工作，但要解决的问题是不同的。这里的情形不是 base class names（基类名字）被 derived class names（派生类名字）隐藏，而是如果我们不告诉它去做，编译器就不会搜索 base class 领域。）

最后一个让你的代码通过编译的办法是显式指定被调用的函数是在 base class（基类）中的：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...
  void sendClearMsg(const MsgInfo& info)
  {
    ...
    MsgSender<Company>::sendClear(info);      // okay, assumes that
    ...                                       // sendClear will be
  }                                           // inherited

  ...
};
```

通常这是一个解决这个问题的最不合人心的方法，因为如果被调用函数是 virtual（虚拟）的，显式限定会关闭 virtual binding（虚拟绑定）行为。

从名字可见性的观点来看，这里每一个方法都做了同样的事情：它向编译器保证任何后继的 base class template（基类模板）的 specializations（特化）都将支持 general template（通用模板）提供的 interface（接口）。所有的编译器在解析一个像 LoggingMsgSender 这样的 derived class template（派生类模板）时，这样一种保证都是必要的，但是如果保证被证实不成立，真相将在后继的编译过程中暴露。例如，如果后面的源代码中包含这些，

```
LoggingMsgSender<CompanyZ> zMsgSender;

MsgInfo msgData;

...                                         // put info in msgData

zMsgSender.sendClearMsg(msgData);           // error! won't compile
```

对 sendClearMsg 的调用将不能编译，因为在此刻，编译器知道 base class（基类）是 template specialization（模板特化）MsgSender<CompanyZ>，它们也知道那个 class（类）没有提供 sendClearMsg 试图调用的 sendClear function（函数）。

从根本上说，问题就是编译器是早些（当 derived class template definitions（派生类模板定义）被解析的时候）诊断对 base class members（基类成员）的非法引用，还是晚些时候（当那些 templates（模板）被特定的 template arguments（模板参数）实例化的时候）再进行。C++ 的方针是宁愿早诊断，而这就是为什么当那些 classes（类）被从 templates（模板）实例化的时候，它假装不知道 base classes（基类）的内容。

Things to Remember

- 在 derived class templates（派生类模板）中，可以经由 "this->" 前缀，经由 using declarations，或经由一个 explicit base class qualification（显式基类限定）引用 base class templates（基类模板）中的名字。

Item 44: 从 templates（模板）中分离出 parameter-independent（参数无关）的代码

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

templates（模板）是节省时间和避免代码重复的极好方法。不必再输入 20 个相似的 classes，每一个包含 15 个 member functions（成员函数），你可以输入一个 class template（类模板），并让编译器实例化出你需要的 20 个 specific classes（特定类）和 300 个函数。（class template（类模板）的 member functions（成员函数）只有被使用时才会被隐式实例化，所以只有在每一个函数都被实际使用时，你才会得到全部 300 个member functions（成员函数）。）function templates（函数模板）也有相似的魅力。不必再写很多函数，你可以写一个 function templates（函数模板）并让编译器做其余的事。这不是很重要的技术吗？

是的，不错……有时。如果你不小心，使用 templates（模板）可能导致 code bloat（代码膨胀）：重复的（或几乎重复的）的代码，数据，或两者都有的二进制码。结果会使源代码看上去紧凑而整洁，但是目标代码臃肿而松散。臃肿而松散很少会成为时尚，所以你需要了解如何避免这样的二进制扩张。

你的主要工具有一个有气势的名字 commonality and variability analysis（通用性与可变性分析），但是关于这个想法并没有什么有气势的东西。即使在你的职业生涯中从来没有使用过模板，你也应该从始至终做这样的分析。

当你写一个函数，而且你意识到这个函数的实现的某些部分和另一个函数的实现本质上是相同的，你会仅仅复制代码吗？当然不。你从这两个函数中分离出通用的代码，放到第三个函数中，并让那两个函数来调用这个新的函数。也就是说，你分析那两个函数以找出那些通用和变化的构件，你把通用的构件移入一个新的函数，并把变化的构件保留在原函数中。类似地，如果你写一个 class，而且你意识到这个 class 的某些构件和另一个 class 的构件是相同的，你不要复制那些通用构件。作为替代，你把通用构件移入一个新的 class 中，然后你使用 inheritance（继承）或 composition（复合）（参见 Items 32，38 和 39）使得原来的 classes 可以访问这些通用特性。原来的 classes 中不同的构件——变化的构件——仍保留在它们原来的位置。

在写 templates（模板）时，你要做同样的分析，而且用同样的方法避免重复，但这里有一个技巧。在 non-template code（非模板代码）中，重复是显式的：你可以看到两个函数或两个类之间存在重复。在 template code（模板代码）中。重复是隐式的：仅有一份 template（模

板）源代码的拷贝，所以你必须培养自己去判断在一个 template（模板）被实例化多次后可能发生的重复。

例如，假设你要为固定大小的 square matrices（正方矩阵）写一个 templates（模板），其中，要支持 matrix inversion（矩阵转置）。

```
template<typename T,           // template for n x n matrices of
         std::size_t n>        // objects of type T; see below for info
class SquareMatrix {           // on the size_t parameter
public:
  ...
  void invert();               // invert the matrix in place
};
```

这个 template（模板）取得一个 type parameter（类型参数）T，但是它还有一个类型为 size_t 的参数——一个 non-type parameter（非类型参数）。non-type parameter（非类型参数）比 type parameter（类型参数）更不通用，但是它们是完全合法的，而且，就像在本例中，它们可以非常自然。

现在考虑以下代码：

```
SquareMatrix<double, 5> sm1;
...
sm1.invert();                  // call SquareMatrix<double, 5>::invert

SquareMatrix<double, 10> sm2;
...
sm2.invert();                  // call SquareMatrix<double, 10>::invert
```

这里将有两个 invert 的拷贝被实例化。这两个函数不是相同的，因为一个作用于 5 x 5 矩阵，而另一个作用于 10 x 10 矩阵，但是除了常数 5 和 10 以外，这两个函数是相同的。这是一个发生 template-induced code bloat（模板导致的代码膨胀）的经典方法。

如果你看到两个函数除了一个版本使用了 5 而另一个使用了 10 之外，对应字符全部相等，你该怎么做呢？你的直觉让你创建一个取得一个值作为一个参数的函数版本，然后用 5 或 10 调用这个参数化的函数以代替复制代码。你的直觉为你提供了很好的方法！以下是一个初步过关的 SquareMatrix 的做法：

```
template<typename T>                    // size-independent base class for
class SquareMatrixBase {                // square matrices
protected:
  ...
  void invert(std::size_t matrixSize); // invert matrix of the given size
  ...
};

template<          typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::invert;   // avoid hiding base version of
                                        // invert; see Item 33
public:
  ...
  void invert() { this->invert(n); }   // make inline call to base class
};                                      // version of invert; see below
                                        // for why "this->" is here
```

就像你能看到的，invert 的参数化版本是在一个 base class（基类）SquareMatrixBase 中的。与 SquareMatrix 一样，SquareMatrixBase 是一个 template（模板），但与 SquareMatrix 不一样的是，它参数化的仅仅是矩阵中的对象的类型，而没有矩阵的大小。因此，所有持有一个给定对象类型的矩阵将共享一个单一的 SquareMatrixBase class。从而，它们共享 invert 在那个 class 中的版本的单一拷贝。

SquareMatrixBase::invert 仅仅是一个计划用于 derived classes（派生类）以避免代码重复的方法，所以它是 protected 的而不是 public 的。调用它的额外成本应该为零，因为 derived classes（派生类）的 inverts 使用 inline functions（内联函数）调用 base class（基类）的版本。（这个 inline 是隐式的——参见 Item 30。）这些函数使用了 "this->" 标记，因为就像 Item 43 解释的，如果不这样，在 templatized base classes（模板化基类）中的函数名（诸如 SquareMatrixBase<T>）被 derived classes（派生类）隐藏。还要注意 SquareMatrix 和 SquareMatrixBase 之间的继承关系是 private 的。这准确地反映了 base class（基类）存在的理由仅仅是简化 derived classes（派生类）的实现的事实，而不是表示 SquareMatrix 和 SquareMatrixBase 之间的一个概念上的 is-a 关系。（关于 private inheritance（私有继承）的信息，参见 Item 39。）

迄今为止，还不错，但是有一个棘手的问题我们还没有提及。SquareMatrixBase::invert 怎样知道应操作什么数据？它从它的参数知道矩阵的大小，但是它怎样知道一个特定矩阵的数据在哪里呢？大概只有 derived class（派生类）才知道这些。derived class（派生类）如何把这些传达给 base class（基类）以便于 base class（基类）能够做这个转置呢？

一种可能是为 SquareMatrixBase::invert 增加另一个的参数，也许是一个指向存储矩阵数据的内存块的开始位置的指针。这样可以工作，但是十有八九，invert 不是 SquareMatrix 中仅有的能被写成一种 size-independent（大小无关）的方式并移入 SquareMatrixBase 的函数。如果有几个这样的函数，全都需要一种找到持有矩阵内的值的内存的方法。我们可以为它们全都增加一个额外的参数，但是我们一再重复地告诉 SquareMatrixBase 同样的信息。这看上去不太正常。

一个可替换方案是让 SquareMatrixBase 存储一个指向矩阵的值的内存区域的指针。而且一旦它存储了这个指针，它同样也可以存储矩阵大小。最后得到的设计大致就像这样：

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T *pMem)     // store matrix size and a
  : size(n), pData(pMem) {}                    // ptr to matrix values

  void setDataPtr(T *ptr) { pData = ptr; }     // reassign pData
  ...

private:
  std::size_t size;                            // size of matrix
  T *pData;                                    // pointer to matrix values
};
```

这样就是让 derived classes（派生类）决定如何分配内存。某些实现可能决定直接在 SquareMatrix object 内部存储矩阵数据：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                               // send matrix size and
  : SquareMatrixBase<T>(n, data) {}            // data ptr to base class
  ...

private:
  T data[n*n];
};
```

这种类型的 objects 不需要 dynamic memory allocation（动态内存分配），但是这些 objects 本身可能会非常大。一个可选方案是将每一个矩阵的数据放到 heap（堆）上：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
  SquareMatrix()                               // set base class data ptr to null,
  : SquareMatrixBase<T>(n, 0),                 // allocate memory for matrix
    pData(new T[n*n])                          // values, save a ptr to the
  { this->setDataPtr(pData.get()); }           // memory, and give a copy of it
  ...                                          // to the base class

private:
  boost::scoped_array<T> pData;                // see Item 13 for info on
};                                             // boost::scoped_array
```

无论数据存储在哪里，从膨胀的观点来看关键的结果在于：现在 SquareMatrix 的许多——也许是全部—— member functions（成员函数）可以简单地 inline 调用它的 base class versions（基类版本），而这个版本是与其它所有持有相同数据类型的矩阵共享的，而无论它们的大小。与此同时，不同大小的 SquareMatrix objects 是截然不同的类型，所以，例如，即使 SquareMatrix<double, 5> 和 SquareMatrix<double, 10> objects 使用

SquareMatrixBase<double> 中同样的 member functions（成员函数），也没有机会将一个 SquareMatrix<double, 5> object 传送给一个期望一个 SquareMatrix<double, 10> 的函数。很好，不是吗？

很好，是的，但不是免费的。将矩阵大小硬性固定在其中的 invert 版本很可能比将大小作为一个函数参数传入或存储在 object 中的共享版本能产生更好的代码。例如，在 size-specific（特定大小）的版本中，sizes（大小）将成为 compile-time constants（编译期常数），因此适用于像 constant propagation 这样的优化，包括将它们作为 immediate operands（立即操作数）嵌入到生成的指令中。在 size-independent version（大小无关版本）中这是不可能做到的。

另一方面，将唯一的 invert 的版本用于多种矩阵大小缩小了可执行码的大小，而且还能缩小程序的 working set（工作区）大小以及改善 instruction cache（指令缓存）中的 locality of reference（引用的局部性）。这些能使程序运行得更快，超额偿还了失去的针对 invert 的 size-specific versions（特定大小版本）的任何优化。哪一个效果更划算？唯一的分辨方法就是在你的特定平台和典型数据集上试验两种方法并观察其行为。

另一个效率考虑关系到 objects 的大小。如果你不小心，将函数的 size-independent 版本（大小无关版本）上移到一个 base class（基类）中会增加每一个 object 的整体大小。例如，在我刚才展示的代码中，即使每一个 derived class（派生类）都已经有了一个取得数据的方法，每一个 SquareMatrix object 都还有一个指向它的数据的指针存在于 SquareMatrixBase class 中，这为每一个 SquareMatrix object 至少增加了一个指针的大小。通过改变设计使这些指针不再必需是有可能的，但是，这又是一桩交易。例如，让 base class（基类）存储一个指向矩阵数据的 protected 指针导致在 Item 22 中描述的封装性的降低。它也可能导致资源管理复杂化：如果 base class（基类）存储了一个指向矩阵数据的指针，但是那些数据既可以是动态分配的也可以是物理地存储于 derived class object（派生类对象）之内的（就像我们看到的），它如何决定这个指针是否应该被删除？这样的问题有答案，但是你越想让它们更加精巧一些，它就会变成更复杂的事情。在某些条件下，少量的代码重复就像是一种解脱。

本 Item 只讨论了由于 non-type template parameters（非类型模板参数）引起的膨胀，但是 type parameters（类型参数）也能导致膨胀。例如，在很多平台上，int 和 long 有相同的二进制表示，所以，可以说，vector<int> 和 vector<long> 的 member functions（成员函数）很可能是相同的——膨胀的恰到好处的解释。某些连接程序会合并同样的函数实现，还有一些不会，而这就意味着在一些环境上一些模板在 int 和 long 上都被实例化而能够引起代码重复。类似地，在大多数平台上，所有的指针类型有相同的二进制表示，所以持有指针类型的模板（例如，list<int*>，list<const int*>，list<SquareMatrix<long, 3>*> 等）应该通常可以使用每一个 member function（成员函数）的单一的底层实现。典型情况下，这意味着与 strongly typed pointers（强类型指针）（也就是 T* 指针）一起工作的 member functions（成员函数）可以通过让它们调用与 untyped pointers（无类型指针）（也就是 void* 指针）一起工作的函数来实现。一些标准 C++ 库的实现对于像 vector，deque 和 list 这样的模板就是这样做的。如果你关心起因于你的模板的代码膨胀，你可能需要用同样的做法开发模板。

Things to Remember

- templates（模板）产生多个 classes 和多个 functions，所以一些不依赖于 template parameter（模板参数）的模板代码会引起膨胀。
- non-type template parameters（非类型模板参数）引起的膨胀常常可以通过用 function parameters（函数参数）或 class data members（类数据成员）替换 template parameters（模板参数）而消除。
- type parameters（类型参数）引起的膨胀可以通过让具有相同的二进制表示的实例化类型共享实现而减少。

Item 45: 用 member function templates（成员函数模板） 接受 "all compatible types"（“所有兼容类型”）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

smart pointers（智能指针）是行为很像指针但是增加了指针没有提供的功能的 objects。例如，Item 13 阐述了标准 auto_ptr 和 tr1::shared_ptr 是怎样被应用于在恰当的时间自动删除的 heap-based resources（基于堆的资源）的。STL containers 内的 iterators（迭代器）几乎始终是 smart pointers（智能指针）；你绝对不能指望用 "++" 将一个 built-in pointer（内建指针）从一个 linked list（线性链表）的一个节点移动到下一个，但是 list::iterators 可以做到。

real pointers（真正的指针）做得很好的一件事是支持 implicit conversions（隐式转换）。derived class pointers（派生类指针）隐式转换到 base class pointers（基类指针），pointers to non-const objects（指向非常量对象的指针）转换到 pointers to const objects（指向常量对象的指针），等等。例如，考虑在一个 three-level hierarchy（三层继承体系）中能发生的一些转换：

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;                  // convert Middle* => Top*
Top *pt2 = new Bottom;                  // convert Bottom* => Top*
const Top *pct2 = pt1;                  // convert Top* => const Top*
```

在 user-defined smart pointer classes（用户定义智能指针类）中模仿这些转换是需要技巧的。我们要让下面的代码能够编译：

```
template<typename T>
class SmartPtr {
public:                               // smart pointers are typically
  explicit SmartPtr(T *realPtr);     // initialized by built-in pointers
  ...
};

SmartPtr<Top> pt1 =                   // convert SmartPtr<Middle> =>
SmartPtr<Middle>(new Middle);         // SmartPtr<Top>

SmartPtr<Top> pt2 =                   // convert SmartPtr<Bottom> =>
SmartPtr<Bottom>(new Bottom);         // SmartPtr<Top>

SmartPtr<const Top> pct2 = pt1;       // convert SmartPtr<Top> =>
                                      // SmartPtr<const Top>
```

在同一个 template（模板）的不同 instantiations（实例化）之间没有 inherent relationship（继承关系），所以编译器认为 SmartPtr<Middle> 和 SmartPtr<Top> 是完全不同的 classes，并不比（比方说）vector<float> 和 Widget 的关系更近。为了得到我们想要的在 SmartPtr classes 之间的转换，我们必须显式地为它们编程。

在上面的 smart pointer（智能指针）的示例代码中，每一个语句创建一个新的 smart pointer object（智能指针对象），所以现在我们就集中于我们如何写 smart pointer constructors（智能指针的构造函数），让它以我们想要的方式运转。一个关键的事实是我们无法写出我们需要的全部 constructors（构造函数）。在上面的 hierarchy（继承体系）中，我们能从一个 SmartPtr<Middle> 或一个 SmartPtr<Bottom> 构造出一个 SmartPtr<Top>，但是如果将来这个 hierarchy（继承体系）被扩充，SmartPtr<Top> objects 还必须能从其它 smart pointer types（智能指针类型）构造出来。例如，如果我们后来加入

```
class BelowBottom: public Bottom { ... };
```

我们就需要支持从 SmartPtr<BelowBottom> objects 到 SmartPtr<Top> objects 的创建，而且我们当然不希望为了做到这一点而必须改变 SmartPtr template。

大体上，我们需要的 constructors（构造函数）的数量是无限的。因为一个 template（模板）能被实例化而产生无数个函数，所以好像我们不需要为 SmartPtr 提供一个 constructor function（构造函数函数），我们需要一个 constructor template（构造函数模板）。这样的 templates（模板）是 member function templates（成员函数模板）（常常被恰如其分地称为 member templates（成员模板））——生成一个 class 的 member functions（成员函数）的 templates（模板）的范例：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>                       // member template
  SmartPtr(const SmartPtr<U>& other);        // for a "generalized
  ...                                        // copy constructor"
};
```

这就是说对于每一种类型 T 和每一种类型 U，都能从一个 SmartPtr<U> 创建出一个 SmartPtr<T>，因为 SmartPtr<T> 有一个取得一个 SmartPtr<U> 参数的 constructor（构造函数）。像这样的 constructor（构造函数）——从一个类型是同一个 template（模板）的不同实例化的 object 创建另一个 object 的 constructor（构造函数）（例如，从一个 SmartPtr<U> 创建一个 SmartPtr<T>）——有时被称为 generalized copy constructors（泛型化拷贝构造函数）。

上面的 generalized copy constructor（泛型化拷贝构造函数）没有被声明为 explicit（显式）的。这是故意为之的。built-in pointer types（内建指针类型）之间的类型转换（例如，从派生类指针到基类指针）是隐式的和不需要 cast（强制转型）的，所以让 smart pointers（智能

指针）模仿这一行为是合理的。在 templatized constructor（模板化构造函数）中省略 explicit 正好做到这一点。

作为声明，SmartPtr 的 generalized copy constructor（泛型化拷贝构造函数）提供的东西比我们想要的还多。是的，我们需要能够从一个 SmartPtr<Bottom> 创建一个 SmartPtr<Top>，但是我们不需要能够从一个 SmartPtr<Top> 创建一个 SmartPtr<Bottom>，这就像颠倒 public inheritance（公有继承）的含义（参见 Item 32）。我们也不需要能够从一个 SmartPtr<double> 创建一个 SmartPtr<int>，因为这和从 int* 到 double* 的 implicit conversion（隐式转换）是不相称的。我们必须设法过滤从这个 member template（成员模板）生成的 member functions（成员函数）的群体。

假如 SmartPtr 跟随 auto_ptr 和 tr1::shared_ptr 的脚步，提供一个返回被这个 smart pointer（智能指针）持有的 built-in pointer（内建指针）的拷贝的 get member function（get 成员函数）（参见 Item 15），我们可以用 constructor template（构造函数模板）的实现将转换限定在我们想要的范围：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)         // initialize this held ptr
  : heldPtr(other.get()) { ... }             // with other's held ptr

  T* get() const { return heldPtr; }
  ...

private:                                     // built-in pointer held
  T *heldPtr;                                // by the SmartPtr
};
```

我们通过 member initialization list（成员初始化列表），用 SmartPtr<U> 持有的类型为 U* 的指针初始化 SmartPtr<T> 的类型为 T* 的 data member（数据成员）。这只有在“存在一个从一个 U* 指针到一个 T* 指针的 implicit conversion（隐式转换）”的条件下才能编译，而这正是我们想要的。最终的效果就是 SmartPtr<T> 现在有一个 generalized copy constructor（泛型化拷贝构造函数），它只有在传入一个 compatible type（兼容类型）的参数时才能编译。

member function templates（成员函数模板）的用途并不限于 constructors（构造函数）。它们的另一个常见的任务是用于支持 assignment（赋值）。例如，TR1 的 shared_ptr（再次参见 Item 13）支持从所有兼容的 built-in pointers（内建指针），tr1::shared_ptrs，auto_ptrs 和 tr1::weak_ptrs（参见 Item 54）构造，以及从除 tr1::weak_ptrs 以外所有这些赋值。这里是从 TR1 规范中摘录出来的一段关于 tr1::shared_ptr 的内容，包括它在声明 template parameters（模板参数）时使用 class 而不是 typename 的偏好。（就像 Item 42 中阐述的，在这里的上下文环境中，它们的含义严格一致。）

```
template<class T> class shared_ptr {
public:
  template<class Y>                                       // construct from
    explicit shared_ptr(Y * p);                           // any compatible
  template<class Y>                                       // built-in pointer,
    shared_ptr(shared_ptr<Y> const& r);                   // shared_ptr,
  template<class Y>                                       // weak_ptr, or
    explicit shared_ptr(weak_ptr<Y> const& r);            // auto_ptr
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);
  template<class Y>                                       // assign from
    shared_ptr& operator=(shared_ptr<Y> const& r);        // any compatible
  template<class Y>                                       // shared_ptr or
    shared_ptr& operator=(auto_ptr<Y>& r);                // auto_ptr
  ...
};
```

除了 generalized copy constructor（泛型化拷贝构造函数），所有这些 constructors（构造函数）都是 explicit（显式）的。这就意味着从 shared_ptr 的一种类型到另一种的 implicit conversion（隐式转换）是被允许的，但是从一个 built-in pointer（内建指针）或其 smart pointer type（智能指针类型）的 implicit conversion（隐式转换）是不被许可的。（explicit conversion（显式转换）——例如，经由一个 cast（强制转型）——还是可以的。）同样引起注意的是 auto_ptrs 被传送给 tr1::shared_ptr 的 constructors（构造函数）和 assignment operators（赋值操作符）的方式没有被声明为 const，于此对照的是 tr1::shared_ptrs 和 tr1::weak_ptrs 的被传送的方式。这是 auto_ptrs 被复制时需要独一无二的被改变的事实的一个必然结果（参见 Item 13）。

member function templates（成员函数模板）是一个极好的东西，但是它们没有改变这个语言的基本规则。Item 5 阐述的编译器可以产生的四个 member functions（成员函数）其中两个是 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）。tr1::shared_ptr 声明了一个 generalized copy constructor（泛型化拷贝构造函数），而且很明显，当类型 T 和 Y 相同时，generalized copy constructor（泛型化拷贝构造函数）就能被实例化而成为 "normal" copy constructor（“常规”拷贝构造函数）。那么，当一个 tr1::shared_ptr object 从另一个相同类型的 tr1::shared_ptr object 构造时，编译器是为 tr1::shared_ptr 生成一个 copy constructor（拷贝构造函数），还是实例化 generalized copy constructor template（泛型化拷贝构造函数模板）？

就像我说过的，member templates（成员模板）不改变语言规则，而且规则规定如果一个 copy constructor（拷贝构造函数）是必需的而你没有声明，将为你自动生成一个。在一个 class 中声明一个 generalized copy constructor（泛型化拷贝构造函数）（一个 member template（成员模板））不会阻止编译器生成它们自己的 copy constructor（拷贝构造函数）（非模板的），所以如果你要全面支配 copy construction（拷贝构造），你必须既声明一个 generalized copy constructor（泛型化拷贝构造函数）又声明一个 "normal" copy constructor（“常规”拷贝构造函数）。这同样适用于 assignment（赋值）。这是从 tr1::shared_ptr 的定义中摘录的一段，可以作为例子：

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                 // copy constructor

  template<class Y>                                // generalized
    shared_ptr(shared_ptr<Y> const& r);            // copy constructor

  shared_ptr& operator=(shared_ptr const& r);      // copy assignment

  template<class Y>                                // generalized
    shared_ptr& operator=(shared_ptr<Y> const& r); // copy assignment
  ...
};
```

Things to Remember

- 使用 member function templates（成员函数模板）生成接受所有兼容类型的函数。
- 如果你为 generalized copy construction（泛型化拷贝构造）或 generalized assignment（泛型化赋值）声明了 member templates（成员模板），你依然需要声明 normal copy constructor（常规拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）。

Item 46: 需要 type conversions（类型转换）时在 templates（模板）内定义 non-member functions（非成员函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 24 阐述了为什么只有 non-member functions（非成员函数）适合于应用到所有 arguments（实参）的 implicit type conversions（隐式类型转换），而且它还作为一个示例使用了一个 Rational class 的 operator* function。我建议你在继续下去之前先熟悉那个示例，因为这个 Item 进行了针对 Item 24 中的示例的一个表面上的无伤大雅的更改（模板化 Rational 和 operator*）的扩展讨论：

```
template<typename T>
class Rational {
public:
  Rational(const T& numerator = 0,     // see Item 20 for why params
           const T& denominator = 1);  // are now passed by reference

  const T numerator() const;           // see Item 28 for why return
  const T denominator() const;         // values are still passed by value,
  ...                                  // Item 3 for why they're const
};

template<typename T>
const Rational<T> operator*(const Rational<T>& lhs,
                            const Rational<T>& rhs)
{ ... }
```

就像在 Item 24 中，我想要支持 mixed-mode arithmetic（混合模式运算），所以我们要让下面这些代码能够编译。我们指望它能，因为我们使用了和 Item 24 中可以工作的代码相同的代码。仅有的区别是 Rational 和 operator* 现在是 templates（模板）：

```
Rational<int> oneHalf(1, 2);          // this example is from Item 24,
                                      // except Rational is now a template

Rational<int> result = oneHalf * 2;   // error! won't compile
```

编译失败的事实暗示对于模板化 Rational 来说，有某些东西和 non-template（非模板）版本不同，而且确实存在。在 Item 24 中，编译器知道我们想要调用什么函数（取得两个 Rationals 的 operator*），但是在这里，编译器不知道我们想要调用哪个函数。作为替代，它们试图 figure out（推断）要从名为 operator* 的 template（模板）中实例化出（也就是创

建）什么函数。它们知道它们假定实例化出的某个名为 operator* 的函数取得两个 Rational<T> 类型的参数，但是为了做这个实例化，它们必须 figure out（推断）T 是什么。问题在于，它们做不到。

在推演 T 的尝试中，它们会察看被传入 operator* 的调用的 arguments（实参）的类型。在当前情况下，类型为 Rational<int>（oneHalf 的类型）和 int（2 的类型）。每一个参数被分别考察。

使用 oneHalf 的推演很简单。operator* 的第一个 parameter（形参）被声明为 Rational<T> 类型，而传入 operator* 的第一个 argument（实参）(oneHalf) 是 Rational<int> 类型，所以 T 一定是 int。不幸的是，对其它参数的推演没那么简单。operator* 的第二个 parameter（形参）被声明为 Rational<T> 类型，但是传入 operator* 的第二个 argument（实参）(2) 的 int 类型。在这种情况下，让编译器如何 figure out（推断）T 是什么呢？你可能期望它们会使用 Rational<int> 的 non-explicit constructor（非显式构造函数）将 2 转换成一个 Rational<int>，这样就使它们推演出 T 是 int，但是它们不这样做。它们不这样做是因为在 template argument deduction（模板实参推演）过程中从不考虑 implicit type conversion functions（隐式类型转换函数）。从不。这样的转换可用于函数调用过程，这没错，但是在你可以调用一个函数之前，你必须知道哪个函数存在。为了知道这些，你必须为相关的 function templates（函数模板）推演出 parameter types（参数类型）（以便你可以实例化出合适的函数）。但是在 template argument deduction（模板实参推演）过程中不考虑经由 constructor（构造函数）调用的 implicit type conversion（隐式类型转换）。Item 24 不包括 templates（模板），所以 template argument deduction（模板实参推演）不是一个问题，现在我们在 C++ 的 template 部分（参见 Item 1），这是主要问题。

在一个 template class（模板类）中的一个 friend declaration（友元声明）可以指涉到一个特定的函数，我们可以利用这一事实为受到 template argument deduction（模板实参推演）挑战的编译器解围。这就意味着 class Rational<T> 可以为 Rational<T> 声明作为一个 friend function（友元函数）的 operator*。class templates（类模板）不依靠 template argument deduction（模板实参推演）（这个过程仅适用于 function templates（函数模板）），所以 T 在 class Rational<T> 被实例化时总是已知的。通过将适当的 operator* 声明为 Rational<T> class 的一个 friend（友元）使其变得容易：

```
template<typename T>
class Rational {
public:
  ...
friend                                              // declare operator*
  const Rational operator*(const Rational& lhs,     // function (see
                           const Rational& rhs);    // below for details)
};

template<typename T>                                // define operator*
const Rational<T> operator*(const Rational<T>& lhs, // functions
                            const Rational<T>& rhs)
{ ... }
```

现在我们对 operator* 的混合模式调用可以编译了，因为当 object oneHalf 被声明为 Rational<int> 类型时，class Rational<int> 被实例化，而作为这一过程的一部分，取得 Rational<int> parameters（形参）的 friend function（友元函数）operator* 被自动声明。作为已声明 function（函数）（并非一个 function template（函数模板）），在调用它的时候编译器可以使用 implicit conversion functions（隐式转换函数）（譬如 Rational 的 non-explicit constructor（非显式构造函数）），而这就是它们如何使得混合模式调用成功的。

唉，在这里的上下文中，“成功”是一个可笑的词，因为尽管代码可以编译，但是不能连接。但是我们过一会儿再处理它，首先我想讨论一下用于在 Rational 内声明 operator* 的语法。

在一个 class template（类模板）内部，template（模板）的名字可以被用做 template（模板）和它的 parameters（参数）的缩写，所以，在 Rational<T> 内部，我们可以只写 Rational 代替 Rational<T>。在本例中这只为我们节省了几个字符，但是当有多个参数或有更长的参数名时，这既能节省击键次数又能使最终的代码显得更清晰。我把这一点提前，是因为 operator* 被声明为取得并返回 Rationals，而不是 Rational<T>s。它就像如下这样声明 operator* 一样合法：

```
template<typename T>
class Rational {
public:
  ...
friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs);
  ...
};
```

然而，使用缩写形式更简单（而且更常用）。

现在返回到连接问题。混合模式代码编译，因为编译器知道我们想要调用一个特定的函数（取得一个 Rational<int> 和一个 Rational<int> 的 operator*），但是那个函数只是在 Rational 内部 declared（被声明），而没有在此处 defined（被定义）。我们打算让 class 之外的 operator* template（模板）提供这个定义，但是这种方法不能工作。如果我们自己声明一个函数（这就是我们在 Rational template（模板）内部所做的事），我们就有责任定义这个函数。当前情况是，我们没有提供定义，这也就是连接器为什么不能找到它。

让它能工作的最简单的方法或许就是将 operator* 的本体合并到它的 declaration（定义）中：

```
template<typename T>
class Rational {
public:
  ...

friend const Rational operator*(const Rational& lhs, const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),       // same impl
                  lhs.denominator() * rhs.denominator());  // as in
}                                                          // Item 24
};
```

确实，这样就可以符合预期地工作：对 operator* 的混合模式调用现在可以编译，连接，并运行。万岁！

关于此技术的一个有趣的观察结论是 friendship 的使用对于访问 class 的 non-public parts（非公有构件）的需求并没有起到什么作用。为了让所有 arguments（实参）的 type conversions（类型转换）成为可能，我们需要一个 non-member function（非成员函数）（Item 24 依然适用）；而为了能自动实例化出适当的函数，我们需要在 class 内部声明这个函数。在一个 class 内部声明一个 non-member function（非成员函数）的唯一方法就是把它做成一个 friend（友元）。那么这就是我们做的。反传统吗？是的。有效吗？毫无疑问。

就像 Item 30 阐述的，定义在一个 class 内部的函数被隐式地声明为 inline（内联），而这也包括像 operator* 这样的 friend functions（友元函数）。你可以让 operator* 不做什么事情，只是调用一个定义在这个 class 之外的 helper function（辅助函数），从而让这样的 inline declarations（内联声明）的影响最小化。在本 Item 的这个示例中，没有特别指出这样做，因为 operator* 已经可以实现为一个 one-line function（单行函数），但是对于更复杂的函数体，这样做也许是合适的。"have the friend call a helper"（"让友元调用辅助函数"）的方法还是值得注意一下的。

Rational 是一个 template（模板）的事实意味着那个 helper function（辅助函数）通常也是一个 template（模板），所以典型情况下在头文件中定义 Rational 的代码看起来大致如下：

```
template<typename T> class Rational;                // declare
                                                    // Rational
                                                    // template
template<typename T>                                   // declare
const Rational<T> doMultiply(const Rational<T>& lhs,   // helper
                             const Rational<T>& rhs);  // template
template<typename T>
class Rational {
public:
  ...

friend
  const Rational<T> operator*(const Rational<T>& lhs,
                              const Rational<T>& rhs)  // Have friend
  { return doMultiply(lhs, rhs); }                     // call helper
  ...
};
```

多数编译器基本上会强迫你把所有的 template definitions（模板定义）都放在头文件中，所以你可能同样需要在你的头文件中定义 doMultiply。（就像 Item 30 阐述的，这样的 templates（模板）不需要 inline（内联）。）可能看起来就像这样：

```
emplate<typename T>                                      // define
const Rational<T> doMultiply(const Rational<T>& lhs,      // helper
                             const Rational<T>& rhs)      // template in
{                                                         // header file,
  return Rational<T>(lhs.numerator() * rhs.numerator(),   // if necessary
                     lhs.denominator() * rhs.denominator());
}
```

当然，作为一个 template（模板），doMultiply 不支持混合模式乘法，但是它不需要。它只被 operator* 调用，而 operator* 支持混合模式运算！本质上，function operator* 支持为了确保被相乘的是两个 Rational objects 而必需的各种 type conversions（类型转换），然后它将这两个 objects 传递给一个 doMultiply template（模板）的适当的实例化来做实际的乘法。配合行动，不是吗？

Things to Remember

- 在写一个 class template（类模板），而这个 class template（类模板）提供了一些 函数，这些函数指涉到支持所有 parameters（参数）的 implicit type conversions（隐式类型转换）的 template（模板）的时候，把这些函数定义为 class template（类模板）内部的 friends（友元）。

Item 47: 为类型信息使用 traits classes（特征类）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

STL 主要是由 containers（容器），iterators（迭代器）和 algorithms（算法）的 templates（模板）构成的，但是也有几个 utility templates（实用模板）。其中一个被称为 advance。advance 将一个指定的 iterator（迭代器）移动一个指定的距离：

```
template<typename IterT, typename DistT>       // move iter d units
void advance(IterT& iter, DistT d);            // forward; if d < 0,
                                               // move iter backward
```

在概念上，advance 仅仅是在做 iter += d，但是 advance 不能这样实现，因为只有 random access iterators（随机访问迭代器）支持 += operation。不够强力的 iterator（迭代器）类型不得不通过反复利用 ++ 或 -- d 次来实现 advance。

嗯，你不记得 STL iterator categories（迭代器种类）了吗？没问题，我们这就做一个简单回顾。对应于它们所支持的操作，共有五种 iterators（迭代器）。input iterators（输入迭代器）只能向前移动，每次只能移动一步，只能读它们指向的东西，而且只能读一次。它们以一个输入文件中的 read pointer（读指针）为原型；C++ 库中的 istream_iterators 就是这一种类的典型代表。output iterators（输出迭代器）与此类似，只不过用于输出：它们只能向前移动，每次只能移动一步，只能写它们指向的东西，而且只能写一次。它们以一个输出文件中的 write pointer（写指针）为原型；ostream_iterators 是这一种类的典型代表。这是两个最不强力的 iterator categories（迭代器种类）。因为 input（输入）和 output iterators（输出迭代器）只能向前移动而且只能读或者写它们指向的地方最多一次，它们只适合 one-pass 运算。

一个更强力一些的 iterator category（迭代器种类）是 forward iterators（前向迭代器）。这种 iterators（迭代器）能做 input（输入）和 output iterators（输出迭代器）可以做到的每一件事情，再加上它们可以读或者写它们指向的东西一次以上。这就使得它们可用于 multi-pass 运算。STL 没有提供 singly linked list（单向链表），但某些库提供了（通常被称为 slist），而这种 containers（容器）的 iterators（迭代器）就是 forward iterators（前向迭代器）。TR1 的 hashed containers（哈希容器）（参见 Item 54）的 iterators（迭代器）也可以属于 forward category（前向迭代器）。

bidirectional iterators（双向迭代器）为 forward iterators（前向迭代器）加上了和向前一样的向后移动的能力。STL 的 list 的 iterators（迭代器）属于这一种类，set，multiset，map 和 multimap 的 iterators（迭代器）也一样。

最强力的 iterator category（迭代器种类）是 random access iterators（随机访问迭代器）。这种 iterators（迭代器）为 bidirectional iterators（双向迭代器）加上了 "iterator arithmetic"（“迭代器运算”）的能力，也就是说，在常量时间里向前或者向后跳转一个任意的距离。这样的运算类似于指针运算，这并不会让人感到惊讶，因为 random access iterators（随机访问迭代器）就是以 built-in pointers（内建指针）为原型的，而 built-in pointers（内建指针）可以和 random access iterators（随机访问迭代器）有同样的行为。vector，deque 和 string 的 iterators（迭代器）是 random access iterators（随机访问迭代器）。

对于五种 iterator categories（迭代器种类）中的每一种，C++ 都有一个用于识别它的 "tag struct"（“标签结构体”）在标准库中：

```
struct input_iterator_tag {};

struct output_iterator_tag {};

struct forward_iterator_tag: public input_iterator_tag {};

struct bidirectional_iterator_tag: public forward_iterator_tag {};

struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

这些结构体之间的 inheritance relationships（继承关系）是正当的 is-a 关系（参见 Item 32）：所有的 forward iterators（前向迭代器）也是 input iterators（输入迭代器），等等，这都是成立的。我们不久就会看到这个 inheritance（继承）的功用。

但是返回到 advance。对于不同的 iterator（迭代器）能力，实现 advance 的一个方法是使用反复增加或减少 iterator（迭代器）的循环的 lowest-common-denominator（最小共通特性）策略。然而，这个方法要花费 linear time（线性时间）。random access iterators（随机访问迭代器）支持 constant-time iterator arithmetic（常量时间迭代器运算），当它出现的时候我们最好能利用这种能力。

我们真正想做的就是大致像这样实现 advance：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
     iter += d;                                      // use iterator arithmetic
  }                                                  // for random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }              // use iterative calls to
    else { while (d++) --iter; }                     // ++ or -- for other
  }                                                  // iterator categories
}
```

这就需要能够确定 iter 是否是一个 random access iterators（随机访问迭代器），依次下来，就需要知道它的类型，IterT，是否是一个 random access iterators（随机访问迭代器）类型。换句话说，我们需要得到关于一个类型的某些信息。这就是 traits 让你做到的：它们允许你在编译过程中得到关于一个类型的信息。

traits 不是 C++ 中的一个关键字或预定义结构；它们是一项技术和 C++ 程序员遵守的惯例。建立这项技术的要求之一是它在 built-in types（内建类型）上必须和在 user-defined types（用户定义类型）上一样有效。例如，如果 advance 被一个指针（譬如一个 const char*）和一个 int 调用，advance 必须有效，但是这就意味着 traits 技术必须适用于像指针这样的 built-in types（内建类型）。

traits 对 built-in types（内建类型）必须有效的事实意味着将信息嵌入到类型内部是不可以的，因为没有办法将信息嵌入指针内部。那么，一个类型的 traits 信息，必须在类型外部。标准的方法是将它放到 template（模板）以及这个 template（模板）的一个或更多的 specializations（特化）中。对于 iterators（迭代器），标准库中 template（模板）被称为 iterator_traits：

```
template<typename IterT>          // template for information about
struct iterator_traits;           // iterator types
```

就像你能看到的，iterator_traits 是一个 struct（结构体）。根据惯例，traits 总是被实现为 struct（结构体）。另一个惯例就是用来实现 traits 的 structs（结构体）以 traits classes（这可不是我捏造的）闻名。

iterator_traits 的工作方法是对于每一个 IterT 类型，在 struct（结构体）iterator_traits<IterT> 中声明一个名为 iterator_category 的 typedef。这个 typedef 被看成是 IterT 的 iterator category（迭代器种类）。

iterator_traits 通过两部分实现这一点。首先，它强制要求任何 user-defined iterator（用户定义迭代器）类型必须包含一个名为 iterator_category 的嵌套 typedef 用以识别适合的 tag struct（标签结构体）。例如，deque 的 iterators（迭代器）是随机访问的，所以一个 deque iterators 的 class 看起来就像这样：

```
template < ... >                    // template params elided
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};
```

然而，list 的 iterators（迭代器）是双向的，所以它们是这样做的：

```
template < ... >
class list {
public:
  class iterator {
  public:
    typedef bidirectional_iterator_tag iterator_category;
    ...
  };
  ...
};
```

iterator_traits 仅仅是简单地模仿了 iterator class 的嵌套 typedef：

```
// the iterator_category for type IterT is whatever IterT says it is;
// see Item 42 for info on the use of "typedef typename"
template<typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator_category iterator_category;
  ...
};
```

这样对于 user-defined types（用户定义类型）能很好地运转。但是对于本身是 pointers（指针）的 iterators（迭代器）根本不起作用，因为不存在类似于带有一个嵌套 typedef 的指针的东西。iterator_traits 实现的第二个部分处理本身是 pointers（指针）的 iterators（迭代器）。

为了支持这样的 iterators（迭代器），iterator_traits 为 pointer types（指针类型）提供了一个 partial template specialization（部分模板特化）。pointers 的行为类似 random access iterators（随机访问迭代器），所以这就是 iterator_traits 为它们指定的种类：

```
template<typename IterT>               // partial template specialization
struct iterator_traits<IterT*>         // for built-in pointer types

{
  typedef random_access_iterator_tag iterator_category;
  ...
};
```

到此为止，你了解了如何设计和实现一个 traits class：

- 识别你想让它可用的关于类型的一些信息（例如，对于 iterators（迭代器）来说，就是它们的 iterator category（迭代器种类））。
- 选择一个名字标识这个信息（例如，iterator_category）。
- 提供一个 template（模板）和一系列 specializations（特化）（例如，iterator_traits），它们包含你要支持的类型的信息。

给出了 iterator_traits ——实际上是 std::iterator_traits，因为它是 C++ 标准库的一部分——我们就可以改善我们的 advance 伪代码：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
     typeid(std::random_access_iterator_tag))
  ...
}
```

这个虽然看起来有点希望，但它不是我们想要的。在某种状态下，它会导致编译问题，但是我们到 Item 48 再来研究它，现在，有一个更基础的问题要讨论。IterT 的类型在编译期间是已知的，所以 iterator_traits<IterT>::iterator_category 可以在编译期间被确定。但是 if 语句还是要到运行时才能被求值。为什么要到运行时才做我们在编译期间就能做的事情呢？它浪费了时间（严格意义上的），而且使我们的执行码膨胀。

我们真正想要的是一个针对在编译期间被鉴别的类型的 conditional construct（条件结构）（也就是说，一个 if...else 语句）。碰巧的是，C++ 已经有了一个得到这种行为的方法。它被称为 overloading（重载）。

当你重载某个函数 f 时，你为不同的 overloads（重载）指定不同的 parameter types（形参类型）。当你调用 f 时，编译器会根据被传递的 arguments（实参）挑出最佳的 overload（重载）。基本上，编译器会说："如果这个 overload（重载）与被传递的东西是最佳匹配的话，就调用这个 f；如果另一个 overload（重载）是最佳匹配，就调用它；如果第三个 overload（重载）是最佳的，就调用它"等等。看到了吗？一个针对类型的 compile-time conditional construct（编译时条件结构）。为了让 advance 拥有我们想要的行为方式，我们必须要做的全部就是创建一个包含 advance 的"内容"的重载函数的多个版本（此处原文有误，根据作者网站勘误修改——译者注），声明它们取得不同 iterator_category object 的类型。我为这些函数使用名字 doAdvance：

```
template<typename IterT, typename DistT>              // use this impl for
void doAdvance(IterT& iter, DistT d,                  // random access
               std::random_access_iterator_tag)       // iterators

{
  iter += d;
}

template<typename IterT, typename DistT>              // use this impl for
void doAdvance(IterT& iter, DistT d,                  // bidirectional
               std::bidirectional_iterator_tag)       // iterators
{
  if (d >= 0) { while (d--) ++iter; }
  else { while (d++) --iter; }
}

template<typename IterT, typename DistT>              // use this impl for
void doAdvance(IterT& iter, DistT d,                  // input iterators
               std::input_iterator_tag)
{
  if (d < 0 ) {
     throw std::out_of_range("Negative distance");    // see below
  }
  while (d--) ++iter;
}
```

因为 forward_iterator_tag 从 input_iterator_tag 继承而来，针对 input_iterator_tag 的 doAdvance 版本也将处理 forward iterators（前向迭代器）。这就是在不同的 iterator_tag structs 之间继承的动机。（实际上，这是所有 public inheritance（公有继承）的动机的一部分：使针对 base class types（基类类型）写的代码也能对 derived class types（派生类类型）起作用。）

advance 的规范对于 random access（随机访问）和 bidirectional iterators（双向迭代器）允许正的和负的移动距离，但是如果你试图移动一个 forward（前向）或 input iterator（输入迭代器）一个负的距离，则行为是未定义的。在我检查过的实现中简单地假设 d 是非负的，因而如果一个负的距离被传入，则进入一个直到计数降为零的非常长的循环。在上面的代码中，我展示了改为一个异常被抛出。这两种实现都是正确的。未定义行为的诅咒是：你无法预知会发生什么。

给出针对 doAdvance 的各种重载，advance 需要做的全部就是调用它们，传递一个适当的 iterator category（迭代器种类）类型的额外 object 以便编译器利用 overloading resolution（重载解析）来调用正确的实现：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  doAdvance(                                              // call the version
    iter, d,                                              // of doAdvance
    typename                                              // that is
      std::iterator_traits<IterT>::iterator_category()    // appropriate for
  );                                                      // iter's iterator
}
```

我们现在能够概述如何使用一个 traits class 了：

- 创建一套重载的 "worker" functions（函数）或者 function templates（函数模板）（例如，doAdvance），它们在一个 traits parameter（形参）上不同。与传递的 traits 信息一致地实现每一个函数。
- 创建一个 "master" function（函数）或者 function templates（函数模板）（例如，advance）调用这些 workers，传递通过一个 traits class 提供的信息。

traits 广泛地用于标准库中。有 iterator_traits，当然，再加上 iterator_category，提供了关于 iterators（迭代器）的四块其它信息（其中最常用的是 value_type —— Item 42 展示了使用它的示例）。还有 char_traits 持有关于 character types（字符类型）的信息，还有 numeric_limits 提供关于 numeric types（数值类型）的信息，例如，可表示值的最小值和最大值，等等。（名字 numeric_limits 令人有些奇怪，因为关于 traits classes 更常用的惯例是以 "traits" 结束，但是它就是被叫做 numeric_limits，所以 numeric_limits 就是我们用的名字。）

TR1（参见 Item 54）引入了一大批新的 traits classes 提供关于类型的信息，包括 is_fundamental<T>（T 是否是一个 built-in type（内建类型）），is_array<T>（T 是否是一个 array type（数组类型）），以及 is_base_of<T1, T2>（T1 是否和 T2 相同或者是 T2 的一

个 base class（基类））。合计起来，TR1 在标准 C++ 中加入了超过 50 个 traits classes。

Things to Remember

- traits classes 使关于类型的信息在编译期间可用。它们使用 templates（模板）和 template specializations（模板特化）实现。
- 结合 overloading（重载），traits classes 使得执行编译期类型 if...else 检验成为可能。

Item 48: 感受 template metaprogramming（模板元编程）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

template metaprogramming (TMP)（模板元编程）是写 template-based（基于模板）的运行于编译期间的 C++ 程序的过程。考虑一下：一个 template metaprogram（模板元程序）是用 C++ 写的运行于 C++ 编译器中的程序。当一个 TMP 程序运行完成，它的输出——从 templates（模板）实例化出的 C++ 源代码片断——随后被正常编译。

如果你仅把它看作古怪的特性而没有打动你，那你就不会对它有足够的深入的思考。

C++ 并不是为 template metaprogramming（模板元编程）设计的，但是自从 TMP 在 1990 年代早期被发现以来，它已被证明非常有用，使 TMP 变容易的扩展很可能会被加入到语言和它的标准库之中。是的，TMP 是被发现，而不是被发明。TMP 所基于的特性在 templates（模板）被加入 C++ 的时候就已经被引进了。所需要的全部就是有人注意到它们能够以一种精巧的而且意想不到的方式被使用。

TMP 有两个强大的力量。首先，它使得用其它方法很难或不可能的一些事情变得容易。第二，因为 template metaprograms（模板元程序）在 C++ 编译期间执行，它们能将工作从运行时转移到编译时。一个结果就是通常在运行时才能被察觉的错误能够在编译期间被发现。另一个结果是使用了 TMP 的 C++ 程序在以下几乎每一个方面都可能更有效率：更小的可执行代码，更短的运行时间，更少的内存需求。（然而，将工作从运行时转移到编译时的一个结果就是编译过程变得更长。使用 TMP 的程序可能比它们的 non-TMP 对等物占用长得多的编译时间。）

考虑 228 页引入的 STL 的 advance 的伪代码。（在 Item 47。你现在可能需要读那个 Item，因为在本 Item 中，我假设你已经熟悉了那个 Item 的内容。）就像 228 页，我突出表示代码中的伪代码部分：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (iter is a random access iterator) {
    iter += d;                                     // use iterator arithmetic
  }                                                // for random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }            // use iterative calls to
    else { while (d++) --iter; }                   // ++ or -- for other
  }                                                // iterator categories
}
```

我们可以用 typeid 把伪代码变成真正的代码。这就产生了一个解决此问题的“常规”的 C++ 方法——它的全部工作都在运行时做：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
  if (typeid(typename std::iterator_traits<IterT>::iterator_category) ==
      typeid(std::random_access_iterator_tag)) {

     iter += d;                                     // use iterator arithmetic
  }                                                 // for random access iters
  else {
    if (d >= 0) { while (d--) ++iter; }             // use iterative calls to
    else { while (d++) --iter; }                    // ++ or -- for other
  }                                                 // iterator categories
}
```

Item 47 指出这个 typeid-based（基于 typeid）的方法比使用 traits 的方法效率低，因为这个方法，（1）类型检测发生在运行时而不是编译期，（2）用来做运行时类型检测的代码必须出现在可执行代码中。实际上，这个例子展示了 TMP 如何能比一个“常规”C++ 程序更高效，因为 traits 方法是 TMP。记住，traits 允许编译时在类型上的 if...else 计算。

我先前谈及一些事情在 TMP 中比在“常规”C++ 中更简单，而 advance 提供了这方面的一个例子。Item 47 提到 advance 的 typeid-based（基于 typeid）的实现可能会导致编译问题，而这就是一个产生问题的例子：

```
std::list<int>::iterator iter;

...

advance(iter, 10);                          // move iter 10 elements forward;
                                            // won't compile with above impl.
```

考虑 advance 为上面这个调用生成的版本。用 iter 和 10 的类型取代 template parameters（模板参数）IterT 和 DistT 之后，我们得到这个：

```
void advance(std::list<int>::iterator& iter, int d)
{
  if (typeid(std::iterator_traits<std::list<int>::iterator>::iterator_category) ==
      typeid(std::random_access_iterator_tag)) {

    iter += d;                                        // error!
  }
  else {
    if (d >= 0) { while (d--) ++iter; }
    else { while (d++) --iter; }
  }
}
```

问题在突出显示的行，使用了 += 的那行。在当前情况下，我们试图在一个 list<int>::iterator 上使用 +=，但是 list<int>::iterator 是一个 bidirectional iterator（双向迭代器）（参见 Item 47），所以它不支持 +=。只有 random access iterators（随机访问迭代器）才支持 +=。此时，我们知道我们永远也不会试图执行那个 += 行，因为那个 typeid 检测对于

list<int>::iterators 永远不成立，但是编译器被责成确保所有源代码是正确的，即使它不被执行，而当 iter 不是一个 random access iterator（随机访问迭代器）时 "iter += d" 是不正确的。traits-based（基于 traits）的 TMP 解决方案与此对比，那里针对不同类型的代码被分离到单独的函数中，其中每一个都只使用了可用于它所针对的类型的操作。

TMP 已经被证明是 Turing-complete（图灵完备）的，这意味着它强大得足以计算任何东西。使用 TMP，你可以声明变量，执行循环，编写和调用函数，等等。但是这些结构看起来与其在“常规”C++ 中的样子非常不同。例如，Item 47 展示了 if...else 条件在 TMP 中是如何通过 templates（模板）和 template specializations（模板特化）被表达的。但那是 assembly-level（汇编层次）的 TMP。针对 TMP 的库（例如 Boost 的 MPL ——参见 Item 55）提供了一种更高层次的语法，虽然还不至于让你把它误认为是“常规”C++。

为了一窥其它东西在 TMP 中如何工作，让我们来看看 loops（循环）。TMP 中没有真正的 looping construct（循环结构），因此 loops（循环）的效果是通过 recursion（递归）完成的。（如果你对 recursion（递归）感到不舒服，在你斗胆进入 TMP 之前一定要解决它。TMP 很大程度上是一个 functional language（函数性语言），而 recursion（递归）之于 functional language（函数性语言）就像电视之于美国流行文化：是密不可分的。）然而，甚至 recursion（递归）都不是常规样式的，因为 TMP loops 不涉及 recursive function calls（递归函数调用），它们涉及 recursive template instantiations（递归模板实例化）。

TMP 的 "hello world" 程序在编译期间计算一个阶乘。它不是一个很令人兴奋的程序，不过，即使不是 "hello world"，也有助于语言入门。TMP 阶乘计算示范了通过 recursive template instantiation（递归模板实例化）实现循环。它也示范了在 TMP 中创建和使用变量的一种方法。看：

```
template<unsigned n>                    // general case: the value of
struct Factorial {                      // Factorial<n> is n times the value
                                        // of Factorial<n-1>

  enum { value = n * Factorial<n-1>::value };

};

template<>                              // special case: the value of
struct Factorial<0> {                   // Factorial<0> is 1
  enum { value = 1 };

};
```

给出这个 template metaprogram（模板元程序）（实际上只是单独的 template metafunction（模板元函数）Factorial），你可以通过引用 Factorial<n>::value 得到 factorial(n) 的值。

代码的循环部分出现在 template instantiation（模板实例化）Factorial<n> 引用 template instantiation（模板实例化）Factorial<n-1> 的地方。就像所有正确的 recursion（递归）有一个导致递归结束的特殊情况。这里，它就是 template specialization（模板特化）Factorial<0>。

Factorial template 的每一个 instantiation（实例化）都是一个 struct，而每一个 struct 都使用 enum hack（参见 Item 2）声明了一个名为 value 的 TMP 变量。value 用于持有阶乘计算的当前值。如果 TMP 有一个真正的循环结构，value 会在每次循环时更新。因为 TMP 在循环的位置使用 recursive template instantiation（递归模板实例化），每一个 instantiation（实例化）得到它自己的 value 的拷贝，而每一个拷贝拥有适合于它在“循环”中所处的位置的值。

你可以像这样使用 Factorial：

```
int main()
{
  std::cout << Factorial<5>::value;            // prints 120
  std::cout << Factorial<10>::value;           // prints 3628800
}
```

如果你觉得这比吃了冰淇淋还凉快，你就具有了一个 template metaprogrammer（模板元程序员）应有的素质。如果 templates（模板）以及 specializations（特化）以及 recursive instantiations（递归实例化）以及 enum hacks 以及对类似 Factorial<n-1>::value 这样的类型的需要使你毛骨悚然，好吧，你是一个不错的常规 C++ 程序员。

当然，Factorial 示范的 TMP 的效用大约就像 "hello world" 示范的任何常规编程语言的效用一样。为了领会为什么 TMP 值得了解，更好地理解它能做什么是很重要的。这里是三个示例：

- Ensuring dimensional unit correctness（确保计量单位正确性）。在科学和工程应用中，计量单位（例如，质量，距离，时间，等等）被正确组合是基础。例如，将一个代表质量的变量赋值给一个代表速度的变量是一个错误，但是用一个时间变量去除距离变量并将结果赋给一个速度变量就是正确的。使用 TMP，不论计算多么复杂，确保（在编译期间）一个程序中所有计量单位组合都是正确的是有可能的。（这是一个如何用 TMP 进行早期错误诊断的例子。）这个 TMP 的使用的一个有趣的方面是能够支持分数指数。这需要这个分数在编译期间被简化以便于编译器能够确认，例如，单位 time1/2 与单位 time4/8 是相同的。

- Optimizing matrix operations（优化矩阵操作）。Item 21 阐释了一些函数，包括 operator*，必须返回新的 objects，而 Item 44 引入了 SquareMatrix class，所以考虑如下代码：

  ```
  typedef SquareMatrix<double, 10000> BigMatrix;
  BigMatrix m1, m2, m3, m4, m5;               // create matrices and
  ...                                         // give them values

  BigMatrix result = m1 * m2 * m3 * m4 * m5;    // compute their product
  ```

  用“常规”方法计算 result 需要四个临时矩阵的创建，用于每一次调用 operator* 的结果。此外，独立的乘法产生了一个四次循环遍历矩阵元素的序列。使用一种与 TMP 相关的被称为 expression templates（表达式模板）的高级模板技术，完全不改变上面的客户代码的语法，而消除临时对象以及合并循环是有可能的。最终的软件使用更少的内存而且运行速度戏剧性地更快。

- Generating custom design pattern implementations（生成自定义的设计模式实现）。像 Strategy（参见 Item 35），Observer，Visitor 等设计模式能用很多方法实现。使用一种被称为 policy-based design（基于 policy 设计）的 TMP-based（基于 TMP）的技术，使得创建代表独立的设计选择的 templates ("policies") 成为可能，这种 templates 能以任意的方法组合以产生带有自定义行为的模式实现。例如，这种技术经常用于允许几个实现了 smart pointer behavioral（智能指针行为）的 policies 的 templates 生成（在编译期间）数百个不同的 smart pointer（智能指针）类型。将类似设计模式和智能指针这样的编程器件的范围大大地扩展，这项技术是通常所说的 generative programming（产生式编程）的基础。

TMP 并不适合于每一个人。它的语法是不符合直觉的，工具支持也很弱（template metaprograms 的调试器？哈！）作为一个相对晚近才发现的"附属"语言，TMP programming 的规则仍然带有试验性质。然而，通过将工作从运行时转移到编译时所提供的效率提升还是能给人留下深刻的印象，而表达在运行时很难或不可能实现的行为的能力也相当有吸引力。

TMP 的支持程度在不断提升。很可能在 C++ 的下一个版本中将对它提供直接的支持，而且 TR1 已经这样做了（参见 Item 54）。关于这一主题的书籍也即将开始出版（目前，C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond 已经出版——译者注），而 web 上的 TMP 信息也正在保持增长。TMP 也许永远不会成为主流，但是对于某些程序员——特别是库开发者——它几乎必然会成为主料。

Things to Remember

- template metaprogramming（模板元编程）能将工作从运行时转移到编译时，这样就能够更早察觉错误并提高运行时性能。

- TMP 能用于在 policy choices 的组合的基础上生成自定义代码，也能用于避免为特殊类型生成不适当的代码。

Item 49: 了解 new-handler 的行为

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

当 operator new 不能满足一个内存分配请求时，它抛出一个 exception（异常）。很久以前，他返回一个 null pointer（空指针），而一些比较老的编译器还在这样做。你依然能得到以前的行为（在一定程度上），但是我要到这个 Item 的最后再讨论它。

在 operator new 因回应一个无法满足的内存请求而抛出一个 exception 之前，它先调用一个可以由客户指定的被称为 new-handler 的 error-handling function（错误处理函数）。（这并不完全确切，operator new 真正做的事情比这个稍微复杂一些，详细细节在 Item 51 提供。）为了指定 out-of-memory-handling function，客户调用 set_new_handler ——一个在 <new> 中声明的标准库函数：

namespace std {

typedef void (*new_handler)(); new_handler set_new_handler(new_handler p) throw(); }

```
就像你能够看到的，new_handler 是一个指针的 typedef，这个指针指向不取得和返回任何东西的函数，而 set_ne
set_new_handler 的形参是一个指向函数的指针，这个函数是 operator new 无法分配被请求的内存时应该调用的。
你可以像这样使用 set_new_handler：
```

// function to call if operator new can't allocate enough memory void outOfMem() { std::cerr << "Unable to satisfy request for memory/n"; std::abort(); } int main() { std::set_new_handler(outOfMem); int *pBigDataArray = new int[100000000L]; ... }

```
如果 operator new 不能为 100,000,000 个整数分配空间，outOfMem 将被调用，而程序将在发出一个错误信息后
当 operator new 不能满足一个内存请求时，它反复调用 new-handler function 直到它能找到足够的内存。引起
* Make more memory available（使得更多的内存可用）。这可能使得 operator new 中下一次内存分配的尝试
* Install a different new-handler（安装一个不同的 new-handler）。如果当前的 new-handler 不能做到
* Deinstall the new-handler（卸载 new-handler），也就是，将空指针传给 set_new_handler。没有 new
* Throw an exception（抛出一个异常），类型为 bad_alloc 或继承自 bad_alloc 的其它类型。这样的异常不
* Not return（不再返回），典型情况下，调用 abort 或 exit。
这些选择使你在实现 new-handler functions 时拥有极大的弹性。
有时你可能希望根据被分配 object 的不同，用不同的方法处理内存分配的失败：
```

class X { public: static void outOfMemory(); ... }; class Y { public: static void outOfMemory(); ... }; X* p1 = new X; // if allocation is unsuccessful, // call X::outOfMemory

Y* p2 = new Y; // if allocation is unsuccessful, // call Y::outOfMemory

```
C++ 没有对 class-specific new-handlers 的支持，但是它也不需要。你可以自己实现这一行为。你只要让每一个
假设你要为 Widget class 处理内存分配失败。你就必须清楚当 operator new 不能为一个 Widget object 分配
```

class Widget { public: static std::new_handler set_new_handler(std::new_handler p) throw(); static void * operator new(std::size_t size) throw(std::bad_alloc); private: static std::new_handler currentHandler; };

```
static class members（静态类成员）必须在 class 定义外被定义（除非它们是 const 而且是 integral ——参
```

std::new_handler Widget::currentHandler = 0; // init to null in the class // impl. file

```
Widget 中的 set_new_handler 函数会保存传递给它的任何指针，而且会返回前次调用时被保存的任何指针，这也正
```

std::new_handler Widget::set_new_handler(std::new_handler p) throw() { std::new_handler oldHandler = currentHandler; currentHandler = p; return oldHandler; }

```
最终，Widget 的 operator new 将做下面这些事情：
1\. 以 Widget 的 error-handling function 为参数调用 standard set_new_handler。这样将 Widget 的
2\. 调用 global operator new 进行真正的内存分配。如果分配失败，global operator new 调用 Widget 的
3\. 如果 global operator new 能够为一个 Widget object 分配足够的内存，Widget 的 operator new 返
以下就是你如何在 C++ 中表达这所有的事情。我们以 resource-handling class 开始，组成部分中除了基本的 R
```

class NewHandlerHolder { public: explicit NewHandlerHolder(std::new_handler nh) // acquire current :handler(nh) {} // new-handler

~NewHandlerHolder() // release it { std::set_new_handler(handler); } private: std::new_handler handler; // remember it

NewHandlerHolder(const NewHandlerHolder&); // prevent copying NewHandlerHolder& // (see Item 14) operator=(const NewHandlerHolder&); };

```
这使得 Widget 的 operator new 的实现非常简单：
```

void * Widget::operator new(std::size_t size) throw(std::bad_alloc) { NewHandlerHolder // install Widget's h(std::set_new_handler(currentHandler)); // new-handler

return ::operator new(size); // allocate memory // or throw

} // restore global // new-handler

```
Widget 的客户像这样使用它的 new-handling capabilities（处理 new 的能力）：
```

void outOfMem(); // decl. of func. to call if mem. alloc. // for Widget objects fails

Widget::set_new_handler(outOfMem); // set outOfMem as Widget's // new-handling function

Widget *pw1 = new Widget; // if memory allocation // fails, call outOfMem

std::string *ps = new std::string; // if memory allocation fails, // call the global new-handling // function (if there is one)

Widget::set_new_handler(0); // set the Widget-specific // new-handling function to // nothing (i.e., null)

Widget *pw2 = new Widget; // if mem. alloc. fails, throw an // exception immediately. (There is // no new- handling function for // class Widget.)

```
无论 class 是什么，实现这个方案的代码都是一样的，所以在其它地方重用它就是一个合理的目标。使它成为可能的一
这个设计的 base class（基类）部分让 derived classes（派生类）继承它们全都需要的 set_new_handler 和
```

template // "mixin-style" base class for class NewHandlerSupport{ // class-specific set_new_handler public: // support

static std::new_handler set_new_handler(std::new_handler p) throw(); static void * operator new(std::size_t size) throw(std::bad_alloc);

... // other versions of op. new — // see Item 52 private: static std::new_handler currentHandler; };

template std::new_handler NewHandlerSupport::set_new_handler(std::new_handler p) throw() { std::new_handler oldHandler = currentHandler; currentHandler = p; return oldHandler; }

template void* NewHandlerSupport::operator new(std::size_t size) throw(std::bad_alloc) { NewHandlerHolder h(std::set_new_handler(currentHandler)); return ::operator new(size); } // this initializes each currentHandler to null template std::new_handler NewHandlerSupport::currentHandler = 0;

```
有了这个 class template（类模板），为 Widget 增加 set_new_handler 支持就很容易了：Widget 只需要从
```

NewHandlerSupport 继承即可。（可能看起来很奇特，但是下面我将解释更多的细节。）

class Widget: public NewHandlerSupport { ... // as before, but without declarations for }; // set_new_handler or operator new

```
这些就是 Widget 为了提供一个 class-specific set_new_handler 所需要做的全部。
但是也许你依然在为 Widget 从 NewHandlerSupport&lt;Widget&gt; 继承而烦恼。如果是这样，当你注意到 Ne
对于 Widget 从一个把 Widget 当作一个 type parameter（类型参数）的 templatized base class（模板化
在这一点上，我发表了一篇文章建议一个更好的名字叫做 "Do It For Me"，因为当 Widget 从 NewHandlerSuppo
像 NewHandlerSupport 这样的 templates 使得为任何有需要的 class 添加一个 class-specific new-hand
直到 1993 年，C++ 还要求 operator new 不能分配被请求的内存时要返回 null。operator new 现在则被指定
```

class Widget { ... }; Widget *pw1 = new Widget; // throws bad_alloc if // allocation fails

if (pw1 == 0) ... // this test must fail

Widget *pw2 =new (std::nothrow) Widget; // returns 0 if allocation for // the Widget fails

if (pw2 == 0) ... // this test may succeed ```

对于异常，nothrow new 提供了比最初看上去更少的强制保证。在表达式 "new (std::nothrow) Widget" 中，发生了两件事。首先，operator new 的 nothrow 版本被调用来为一个 Widget object 分配足够的内存。如果这个分配失败，众所周知，operator new 返回 null pointer。然而，如果它成功了，Widget constructor 被调用，而在此刻，所有打的赌都失效了。Widget constructor 能做任何它想做的事。它可能自己 new 出来一些内存，而如果它这样做了，它并没有被强迫使用 nothrow new。那么，虽然在 "new (std::nothrow) Widget" 中调用的 operator new 不会抛出，Widget constructor 却可以。如果它这样做了，exception 像往常一样被传播。结论？使用 nothrow new 只能保证 operator new 不会抛出，不能保证一个像 "new (std::nothrow) Widget" 这样的表达式绝不会导致一个 exception。在所有的可能性中，你最好绝不需要 nothrow new。

无论你是使用 "normal"（也就是说，exception-throwing）new，还是它的稍微有些矮小的堂兄弟，理解 new-handler 的行为是很重要的，因为它可以用于两种形式。

Things to Remember

- set_new_handler 允许你指定一个当内存分配请求不能被满足时可以被调用的函数。
- nothrow new 作用有限，因为它仅适用于内存分配，随后的 constructor 调用可能依然会抛出 exceptions。

Item 50: 领会何时替换 new 和 delete 才有意义

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

让我们先回顾一下基础。为什么有些人想要替换编译器提供的 operator new 或 operator delete 版本呢？有三个最主要的原因：

- 为了监测使用错误。对由 new 产生的内存没有实行 delete 会导致内存泄漏。在 new 出的内存上实行多于一次的 delete 会引发未定义行为。如果 operator new 保存一个已分配地址的列表，而 operator delete 从这个列表中移除地址，这样就很容易监测到上述使用错误。同样，某种编程错误会导致 data overruns（数据上溢）（在一个已分配块的末端之后写入）和 underruns（下溢）（在一个已分配块的始端之前写入）。在对于客户可用的内存的之前和之后，自定义 operator news 可以跨越分配块，在这些空间放置已知的字节模式 ("signatures")。operator deletes 会去检查这些 signatures 是否依旧保持原样。如果不是，在这个分配块的生存期间的某个时刻发生了一个上溢或者下溢，而且 operator deletes 可以记录这件事以及那个讨厌的指针的值。

- 为了提升性能。由编译器加载的 operator new 和 operator delete 版本是为了多种用途而设计的。它们必须被长时间运行的程序（例如，web servers），接受，但是，它们也必须被运行时间少于一秒的程序接受。它们必须处理大内存块，小内存块，以及两者混合的请求序列。它们必须适应广泛的分配模式，从存在于整个程序的持续期间的少数几个区块的动态分配到大量短寿命 objects 的持续不断的分配和释放。它们必须为堆碎片化负责，对这个过程，如果不进行控制，最终会导致不能满足对大内存块的请求，即使有足够的自由内存分布在大量小块中。

> 由于内存管理器的特定需求，由编译器加载的 operator news 和 operator deletes 采取了 middle-of-the-road strategy（中间路线策略）不值得大惊小怪。它们的工作对每一个人来说都说得过去，但是对谁都不是最合适的。如果你对你的程序的动态内存的应用模式有充分的理解，你可能经常发现 operator new 和 operator delete 的自定义版本胜于缺省版本。对于“胜于”，我的意思是它们运行更快——有时会有数量级的提升——而且它们需要更少的内存——最高会少于 50%。对于某些（尽管不意味着全部）应用程序，用自定义版本取代普通的 new 和 delete 是获得重大性能提升的一个简单方法。

- 为了收集使用方法的统计数据。在一头扎入编写自定义 news 和 deletes 的道路之前，收集一下你的软件如何使用动态内存的信息还是比较明智的。被分配区块大小的分布如何？生存期的分布如何？它们的分配和释放的顺序是趋向于 FIFO ("first in, first out")（“先进先出”），或者 LIFO ("last in, first out")（“后进先出”）的顺序，还是某种接近于随机的顺序？使用模式会随着时间而变化吗？例如，你的软件是不是在不同的运行阶段有

> 不同的分配/释放模式？在任一时间内使用中的动态分配内存的最大值（也就是说，它的“最高水位”）是多少？operator new 和 operator delete 的自定义版本使得收集这类信息变得容易。

在概念上，编写一个自定义 operator new 相当简单。例如，这是一个便于 under- 和 overruns 的检测的 global operator new 的主要部分。这里有很多小麻烦，但是我们马上就来关注一下它们。

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;

// this code has several flaws—see below
void* operator new(std::size_t size) throw(std::bad_alloc)
{
  using namespace std;

  size_t realSize = size + 2 * sizeof(int);    // increase size of request so2
                                               // signatures will also fit inside

  void *pMem = malloc(realSize);               // call malloc to get theactual
  if (!pMem) throw bad_alloc();                // memory

  // write signature into first and last parts of the memory
  *(static_cast&lt;int*&gt;(pMem)) = signature;
  *(reinterpret_cast&lt;int*&gt;(static_cast&lt;Byte*&gt;(pMem)+realSize-sizeof(int))) =
  signature;

  // return a pointer to the memory just past the first signature
  return static_cast&lt;Byte*&gt;(pMem) + sizeof(int);
}
```

这个 operator new 的大多数缺陷都与它没有遵循叫这个名字的函数的 C++ 惯例有关。例如，Item 51 阐明：所有的 operator new 都应该包含一个调用 new-handling function 的循环，但是这里没有。但是，Item 51 正是专用于这样的惯例，所以我在这里忽略它们。我现在要关注一个更微妙的问题：*alignment*（排列对齐）。

很多计算机架构要求特定类型的数据要放置在内存中具有特定性质的地址中。例如，一种架构可能要求 pointers（指针）要出现在四的倍数的地址上（也就是说，按照四字节对齐）或者 doubles（双精度浮点型）必须出现在八的倍数的地址上（也就是说，按照八字节对齐）。不遵守这样的约束会导致 hardware exceptions at runtime（运行时硬件异常）。其它的架构可能会宽容一些，但是如果满足了排列对齐的次序会得到更好的性能。例如，在 Intel x86 架构上 doubles（双精度浮点型）可以按照任意字节分界排列，但是如果他们按照八字节对齐，访问速度会快得多。

alignment（排列对齐）在这里有重大意义，因为 C++ 要求所有的 operator news 返回适合任何数据类型的排列的指针。malloc 也工作于同样的要求下，所以，让 operator new 返回它从 malloc 得到的指针是安全的。然而，在上面的 operator news 中，我们没有返回我们从 malloc 得到的指针，我们返回的指针比我们从 malloc 得到的指针偏移了一个 int 大小。无法保证这是安全的！如果客户调用 operator new 为一个 double（或者，如果我们正在编写

operator new[]，一个 doubles 的数组）申请足够的内存，而且我们正在运行一台 ints 是四个字节大小而 doubles 需要八字节对齐的机器，我们就可能返回对齐不恰当的指针。这可以导致程序崩溃。或者，它只是导致运行速度变慢。无论哪种情况，这或许都不是我们想要的。

像 alignment（排列对齐）这样的细节可以用于区分专业品质的内存管理器和那些由需要解决其它任务而心烦意乱的程序员匆匆拼凑出来的东西。编写一个几乎能工作的自定义内存管理器相当容易。编写一个工作得很好的要困难得多。作为一个一般规则，我建议你不要致力于此，除非你不得不做。

很多情况下，你并非不得不做。有些编译器提供选项开关用为它们的 memory management functions（内存管理函数）打开调试和记录的功能。快速浏览一下你的编译器的文档也许可以打消你编写 new 和 delete 的念头。在很多平台上，商用产品可以替代随编译器提供的 memory management functions（内存管理函数）。为了利用它们的增强的功能以及（或许会有的）更好的性能，你需要做的全部就是重新链接。（当然，你还必须把它们买回来。）

另一个选择是开源的内存管理器。它们可用于多种平台，所以你可以下载并试用。出自于 Boost（参见 Item 55）的 Pool library 就是一个这样的开源分配器。Pool library 提供了针对自定义内存管理能提供帮助的最通常的情况之一（大数量 small objects（小对象）的分配）进行了调谐的分配器。很多 C++ 书籍，包括本书的早期版本，展示了一个 high-performance small-object allocator（高性能小对象分配器）的代码，但是它们通常忽略了可移植性和排列对齐的考虑以及线程安全等等诸如此类的麻烦的细节。真正的库会注意用健壮得多的代码。即使你决定编写你自己的 news 和 deletes，看一下开源版本很可能会为你提供对“区分几乎起作用和真正起作用”的容易忽略的细节的洞察力。（已知 alignment（排列对齐）就是一个这样的细节，值得一提的是，TR1（参见 Item 54）包含了对已发现的类型特定的排列对齐要求的支持。）

这个 Item 的主题是了解何时替换 new 和 delete 的缺省版本（无论是基于全局的还是 per-class 的）才有意义。我们现在应该比前面更详细地总结一下时机问题。

- 为了监测使用错误（如前）。
- 为了收集有关动态分配内存的使用的统计数据（如前）。
- 为了提升分配和回收的速度。general-purpose allocators（通用目的的分配器）通常（虽然不总是）比自定义版本慢很多，特别是如果自定义版本是为某种特定类型的 objects 专门设计的。class-specific allocators（类专用分配器）是 fixed-size allocators（固定大小分配器）（就像 Boost 的 Pool library 所提供的那些）的一种典范应用。如果你的程序是 single-threaded（单线程）的，而你的编译器缺省的内存管理例程是 thread-safe（线程安全）的，通过编写 thread-unsafe allocators（非线程安全分配器）你可以获得相当的速度提升。当然，在得出 operator new 和 operator delete 对速度提升有价值的结论之前，确实测定你的程序以保证这些函数是真正的瓶颈。

- 为了减少缺省内存管理的空间成本。general-purpose memory managers（通用目的的内存管理器）通常（虽然不总是）不仅比自定义版本慢，而且还经常使用更多的内存。这是因为它们经常为每个已分配区块招致某些成本。针对 small objects（小对象）调谐的分配器（诸如 Boost 的 Pool library 中的那些）从根本上消除了这样的成本。

- 为了调整缺省分配器不适当的排列对齐。就像我前面提到的，在 x86 架构上，当 doubles 按照八字节对齐时访问速度是最快的。哎呀，有些随编译器提供的 operator news 不能保证 doubles 的动态分配按照八字节对齐。在这种情况下，用保证按照八字节对齐的 operator new 替换掉缺省版本，可以使程序性能得到较大提升。

- 为了聚集相关的 **objects**，使它们彼此靠近。如果你知道特定的 data structures（数据结构）通常会在一起使用，而且你想将在这些数据上工作时的页错误频率降到最低，那么为这些 data structures（数据结构）创建一个独立的 heap（堆）以便让它们尽可能地聚集在不多的几个页上就是有意义的。new 和 delete 的 placement versions（参见 Item 52）使得完成这样的聚集成为可能。

- 为了获得不同寻常的行为。有时你想让 operators new 和 delete 做一些编译器装备版本没有提供的事情。例如，你可能想在共享内存中分配和回收区块，但是只能通过一个 C API 来管理那片内存。编写 new 的 delete 的自定义版本（或许是 placement versions ——再次参见 Item 52）允许你用 C++ 衣服来遮住那个 C API。作为另一个例子，你可以编写一个自定义的 operator delete 用 zeros 复写被回收的内存以提高应用程序数据的安全性。

Things to Remember

- 有很多正当的编写 new 和 delete 的自定义版本的理由，包括改进性能，调试 heap（堆）用法错误，以及收集 heap（堆）用法信息。

Item 51: 编写 new 和 delete 时要遵守惯例

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 50 讲解了什么时候你可能需要编写 operator new 和 operator delete 的你自己的版本，但是没有讲解当你这样做时必须遵循的惯例。这些规则并不难以遵循，但有一些不那么直观，所以了解它们是什么非常重要。

我们从 operator new 开始。实现一个符合惯例的 operator new 需要有正确的返回值，在没有足够的内存可用时调用 new-handling function（参见 Item 49），并做好应付无内存请求的准备。你还要避免无意中对 new 的“常规”形式的覆盖，虽然这更多的是一个 class interface（类接口）的问题，而并非是一个实现的需求，它将在 Item 52 讨论。

operator new 的返回值部分很容易。如果你能提供所请求的内存，你就返回一个指向它的指针。如果你不能，你应该遵循 Item 49 描述的规则并抛出一个 bad_alloc 类型的 exception（异常）。

然而，它也不完全那么简单，因为 operator new 实际上不止一次设法分配内存，每次失败后调用 new-handling function。在此假设 new-handling function 能做些事情释放一些内存。只有当指向 new-handling function 的指针为空时，operator new 才抛出一个 exception（异常）。

奇怪的是，C++ 要求即使请求零字节，operator new 也要返回一个合理的指针。（需要这种怪异的行为来简化语言的其它部分。）在这种情况下，一个 non-member（非成员）的 operator new 的伪代码如下：

```
void * operator new(std::size_t size) throw(std::bad_alloc)
{                                      // your operator new might
  using namespace std;                 // take additional params

  if (size == 0) {                     // handle 0-byte requests
    size = 1;                          // by treating them as
  }                                    // 1-byte requests

  while (true) {
   _attempt to allocate size bytes;_
    if (_the allocation was successful_)
       return (_a pointer to the memory_);

    // allocation was unsuccessful; find out what the
    // current new-handling function is (see below)
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);

    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

将零字节请求当作他们真的请求了一个字节来处理的窍门看起来很龌龊，但是它简单，合法，有效，无论如何，你估摸着的请求零字节这种事情发生的频率有多大呢？

你可能在不经意中还看到伪代码中将 new-handling function pointer 设置为空，然后又马上重置为它原来的值。遗憾的是，没有办法直接得到 new-handling function pointer，所以你必须调用 set_new_handler 以得知它是什么。拙劣，的确，但是也有效，至少对 single-threaded（单线程）代码没问题。在 multithreaded（多线程）环境中，你可能需要某种锁以便安全地摆布隐藏在 new-handling function 后面的（全局的）data structures（数据结构）。

Item 49 谈及 operator new 包含一个无限循环，而上面的代码明确地展示了这个循环，"while (true)" 差不多会尽其所能地无限做下去。跳出循环的唯一出路是内存被成功分配或 new-handling function 做了 Item 49 中描述的事情之一：使得更多的内存可用，安装一个不同的 new-handler，卸载 new-handler，抛出一个 bad_alloc 或从 bad_alloc 派生的 exception（异常），或不再返回。现在，new-handler 为什么要做这些事情之一已经很清楚了。如果它不这样做，operator new 内的循环永远不会停止。

很多人没有意识到 operator new member functions（成员函数）会被 derived classes（派生类）继承。这会引起一些有趣的复杂性。在前面的 operator new 伪代码中，注意那个函数设法分配 size 个字节（除非 size 是零）。因为它是传递给这个函数的 argument（实参），所以它有着明确的意义。然而，就像 Item 50 所讲的，编写一个自定义的内存管理器的最常见的原因之一是为了优化某个特定 class 的 objects 的分配，而不是某个 class 或它的任何 derived classes（派生类）的。也就是说，给定一个 class X 的 operator new，这个函数的行为通常是为大小为 sizeof(X) 的 objects 调谐的——绝不会更大或者更小。然而，由于 inheritance（继承），就有可能一个 base class（基类）中的 operator new 被调用来为一个 derived class（派生类）的 object 分配内存：

```
class Base {
public:
  **static void * operator new(std::size_t size) throw(std::bad_alloc)**;
  ...
};

class Derived: public Base                // Derived doesn't declare
{ ... };                                  // operator new

Derived *p = **new Derived**;                 // calls Base::operator new!
```

如果 Base 的 class-specific（类专用）的 operator new 不是被设计成应付这种情况的——它很可能不是——它处理这种局面的最佳办法就是把这个请求“错误”内存量的调用甩给 standard operator new，就像这样：

```
void * Base::operator new(std::size_t size) throw(std::bad_alloc)
{
  **if (size != sizeof(Base))**                  // if size is "wrong,"
     **return ::operator new(size)**;            // have standard operator
                                          // new handle the request

  ...                                     // otherwise handle
                                          // the request here
}
```

“不许动！”我听到你喊，“你忘了检查 size 是零这种 pathological-but-nevertheless-possible（病态然而可能）的情况！”实际上，我没有，还有，当你大声抱怨的时候拜托不要使用连字符。测试依然在那，它只是与 size 和 sizeof(Base) 的比较合在了一起。C++ 工作在一些神秘的方式中，这些方式之一就是强制规定所有的独立 objects 都具有非零的大小（参见 Item 39）。根据定义，sizeof(Base) 绝不会是零，所以如果 size 是零，请求将转发给 ::operator new，而以一种合理的方式处置这个请求就成为那个函数的职责。

如果你想要在每一个 class 的基础上控制数组的内存分配，你需要实现 operator new 的专用于数组的兄弟，operator new[]。（这个 function 通常被叫做 "array new"，因为要确定 "operator new[]" 如何发音实在是太难了。）如果你决定要编写 operator new[]，记住你所做的全部是分配一大块 raw memory（裸内存）——你不能针对还不存在的数组中的 objects 做任何事情。实际上，你甚至不能确定数组中会有多少个 objects。首先，你不知道每个 object 有多大。毕竟，一个 base class（基类）的 operator new[] 通过继承可以被调用来为一个 derived class objects（派生类对象）的数组分配内存，而 derived class objects（派生类对象）通常都比 base class objects（基类对象）更大。

因此，在 Base::operator new[] 中，你不能断定每一个加到数组中的 object 的大小一定是 sizeof(Base)，而这就意味着，你不能断定数组中的 objects 的数量是 (*bytes requested*)/sizeof(Base)。第二，传递给 operator new[] 的 size_t 参数可能比充满 objects 的内存还要大一些，因为，就像 Item 16 讲到的，dynamically allocated arrays（动态分配数组）可能包括额外的空间用于存储数组元素的数量。

编写 operator new 时，你需要遵循的惯例也就到此为止了。对于 operator delete，事情就更简单了，你需要记住的全部大约就是 C++ 保证删除空指针总是安全的，所以你需要遵循这个保证。下面是一个非成员的 operator delete 的伪代码：

```
void operator delete(void *rawMemory) throw()
{
  if (rawMemory == 0) return;            // do nothing if the null
                                         // pointer is being deleted

  _deallocate the memory pointed to by rawMemory;_
}
```

这个函数的成员版本也很简单，只是你必须确保检查被删除东西的大小。假设你的 class-specific（类专用）的 operator new 将“错误”大小的请求转发给 ::operator new，你也可以将“错误大小”的删除请求转发给 ::operator delete：

```
class Base {                            // same as before, but now
public:                                 // operator delete is declared

  static void * operator new(std::size_t size) throw(std::bad_alloc);
  **static void operator delete(void *rawMemory, std::size_t size) throw();**
  ...
};
void Base::operator delete(void *rawMemory, std::size_t size) throw()
{
  if (rawMemory == 0) return;           // check for null pointer

  **if (size != sizeof(Base)) {**           // if size is "wrong,"
     **::operator delete(rawMemory);**      // have standard operator
     **return;**                            // delete handle the request
  **}**
  _deallocate the memory pointed to by rawMemory;_
  return;
}
```

有趣的是，如果被删除的 object 是从一个缺少 virtual destructor（虚拟析构函数）的 base class（基类）派生出来的，C++ 传递给 operator delete 的 size_t 值也许是不正确的。这已经足够作为“确保你的 base classes（基类）拥有 virtual destructors（虚拟析构函数）”的原因了，除此之外，Item 7 描述了另一个，论证得更好的原因。至于当前，简单地记住如果你在 base classes（基类）中遗漏了 virtual destructors（虚拟析构函数），operator delete functions 可能无法正确工作。

Things to Remember

- operator new 应该包含一个设法分配内存的无限循环，如果它不能满足一个内存请求，应该调用 new-handler，还应该处理零字节请求。class-specific（类专用）版本应该处理对比预期更大的区块的请求。
- operator delete 如果收到一个空指针应该什么都不做。class-specific（类专用）版本应该处理比预期更大的区块。

Item 52: 如果编写了 placement new，就要编写 placement delete

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在 C++ 动物园中，placement new 和 placement delete 并不是最常遇到的野兽，所以如果你和它们不熟也不必担心。作为替代，回想一下 Items 16 和 17，当你写下一个这样的 new 表达式，

```
Widget *pw = new Widget;
```

有两个函数会被调用：一个是 operator new 用于分配内存，第二个是 Widget 的 default constructor（缺省构造函数）。

假设第一个调用成功，而第二个调用导致抛出一个 exception（异常）。这种情况下，第 1 步中完成的内存分配必须被撤销。否则就是一个内存泄漏。客户代码不可能回收这些内存，因为，如果 Widget 的 constructor（构造函数）抛出一个 exception（异常），pw 根本就没有被赋值。对于客户来说无法得到指向应该被回收的内存的指针。所以撤销第 1 步的职责必然落在了 C++ runtime system（C++ 运行时系统）的身上。

runtime system（运行时系统）恰当地调用与它在第 1 步中调用的 operator new 的版本相对应的 operator delete，但是只有在它知道哪一个 operator delete——可能有许多——最恰当的时候它才能做到这一点。如果你正在摆弄具有常规的 signatures（识别特征）的 new 和 delete 版本，这不成问题，因为常规的 operator new，

```
void* operator new(std::size_t) throw(std::bad_alloc);
```

对应常规的 operator delete：

```
void operator delete(void *rawMemory) throw();  // normal signature
                                                // at global scope

void operator delete(void *rawMemory,           // typical normal
                     std::size_t size) throw(); // signature at class
                                                // scope
```

当你只使用 new 和 delete 的常规形式时，runtime system（运行时系统）找出知道如何撤销 new 所做的事情的 delete 没什么麻烦。然而，当你开始声明 operator new 的非常规形式——带有额外参数的形式的时候，which-delete-goes-with-this-new（哪一个 delete 和这个 new 配

对）的问题就出现了。

例如，假设你编写了一个 class-specific（类专用）的 operator new，它需要一个用于记录分配信息的 ostream 的规格描述，而你又编写了一个常规的 class-specific（类专用）的 operator delete：

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size,              // non-normal
                            std::ostream& logStream)       // form of new
    throw(std::bad_alloc);

  static void operator delete(void *pMemory                // normal class-
                              std::size_t size) throw();   // specific form
                                                           // of delete
  ...
};
```

这个设计是成问题的，但是在我们探究为什么之前，我们需要做一个简要的术语说明。

当一个 operator new function 持有额外的参数（除了那个必要的 size*t 参数），这个 function* 就被称为 *new* 的 _*placement* 版本。前面那个 operator new 就是这样一个 placement 版本。有一个特别有用的 placement new，它持有一个指针，这个指针指定了一个 object 被构造的位置。那个 operator new 如下：

```
void* operator new(std::size_t, **void *pMemory**) throw();   // "placement
                                                            // new"
```

new 的这个版本是 C++ 标准库的一部分，只要 #include <new> 你就可以访问它。需要指出，这个 new 用于 vector 内部，在 vector 的尚未使用的空间内创建 objects。它也是最初的 placement new。实际上，这就是这类函数被称为 *placement new* 的来历。这就意味着术语 "placement new" 被赋予了更多的含义。大多数情况下，当人们谈到 placement new，他们谈的就是这个特定的函数，持有一个 void* 类型的额外参数的 operator new。较少情况下，他们谈的是持有额外参数的 operator new 的任意版本。根据上下文通常可以搞清楚任何暧昧，重要的是要了解到通用术语 "placement new" 意味着持有额外参数的 new 的任意版本，因为短语 "placement delete"（过一会儿我们就会遇到它）直接起源于它。

我们让我们先返回到 Widget class 的 declaration（声明），就是我说设计成问题的那个。麻烦就在于这个 class 会引发微妙的 memory leaks（内存泄漏）。考虑如下客户代码，在动态创建一个 Widget 时，它将在 cerr 记录分配信息：

```
Widget *pw = new (std::cerr) Widget; // call operator new, passing cerr as
                                     // the ostream; _this leaks memory_
                                     // _if the Widget constructor throws_
```

重申一次，如果内存分配成功而 Widget constructor（构造函数）抛出一个 exception（异常），runtime system（运行时系统）有责任撤销 operator new 所执行的分配。然而，runtime system（运行时系统）不能真正了解被调用的 operator new 版本是如何工作的，所以它自己无法撤销那个分配。runtime system（运行时系统）转而寻找一个和 operator new 持有相同数量和类型额外参数的 operator delete 版本，而且，如果它找到了，它将调用它。在当前情况下，operator new 持有一个 ostream& 类型的额外参数，所以相应的 operator delete 应该具有这样的 signature（识别特征）：

```
void operator delete(void *, **std::ostream&**) throw();
```

与 new 的 placement 版本类似，持有额外参数的 operator delete 版本被称为 *placement deletes*。当前情况下，Widget 没有声明 operator delete 的 placement 版本，所以 runtime system（运行时系统）不知道如何撤销所调用的 placement new 所做的事情。结果，它什么都不做。在本例中，如果 Widget constructor（构造函数）抛出一个 exception（异常），没有 *operator delete* 可以被调用！

规则很简单：如果一个带有额外参数的 operator new 没有带有同样额外参数的 operator delete 相匹配，当一个由 new 生成的内存分配需要撤销的时候没有 operator delete 可以被调用。为了消除前面的代码中的 memory leak（内存泄漏），Widget 需要声明一个与 logging placement new 相对应的 placement delete：

```
class Widget {
public:
  ...
  static void* operator new(std::size_t size, std::ostream& logStream)
    throw(std::bad_alloc);
  static void operator delete(void *pMemory) throw();

  **static void operator delete(void *pMemory, std::ostream& logStream)**
    **throw();**
  ...
};
```

这样改变之后，如果从下面这个语句的 Widget constructor（构造函数）中抛出一个 exception（异常），

```
Widget *pw = new (std::cerr) Widget;   // as before, but no leak this time
```

相应的 placement delete 自动被调用，而这就让 Widget 确保没有内存被泄漏。

然而，考虑以下情况会发生什么，如果没有抛出 exception（异常）（这是通常的情况）而我们的客户代码中又有一个 delete：

```
delete pw;                            // invokes the normal
                                      // operator delete
```

就像注释中所说的，这样将调用常规 operator delete，而不是 placement 版本。只有在调用一个与 placement new 相关联的 constructor（构造函数）时发生一个 exception（异常），placement delete 才会被调用。将 delete 施加于一个指针（诸如上面的 pw）绝对不会引起一个 delete 的 placement 版本的调用。绝对不会。

这就意味着为了预防所有与 new 的 placement 版本相关的 memory leaks（内存泄漏），你必须既提供常规 operator delete（用于构造过程中没有抛出 exception（异常）时），又要提供一个持有与 operator new 相同的 extra arguments（额外参数）的 placement 版本（用于相反情况）。这样，你就再也不会因为微妙的 memory leaks（内存泄漏）而睡不着觉了。好吧，至少是不会因为这里这些微妙的 memory leaks（内存泄漏）。

顺便说一下，因为 member function（成员函数）的名字会覆盖外围的具有相同名字的函数（参见 Item 33），你需要小心避免用 class-specific（类专用）的 news 覆盖你的客户所希望看到的其它 news（包括其常规版本）。例如，如果你有一个只声明了一个 operator new 的 placement 版本的 base class（基类），客户将发现 new 的常规形式对他们来说无法使用：

```
class Base {
public:
  ...

  static void* operator new(std::size_t size,           // this new hides
                            std::ostream& logStream)    // the normal
    throw(std::bad_alloc);                              // global forms
  ...
};
Base *pb = new Base;                        // error! the normal form of
                                            // operator new is hidden

Base *pb = new (std::cerr) Base;            // fine, calls Base's
                                            // placement new
```

同样，derived classes（派生类）中的 operator news 覆盖 operator news 的全局和继承来的版本的 operator new：

```
class Derived: public Base {                   // inherits from Base above
public:
  ...

  static void* operator new(std::size_t size)  // redeclares the normal
      throw(std::bad_alloc);                   // form of new
  ...
};
Derived *pd = new (std::clog) Derived;         // error! Base's placement
                                               // new is hidden

Derived *pd = new Derived;                     // fine, calls Derived's
                                               // operator new
```

Item 33 讨论了这种名字覆盖的需要考虑的细节，如果打算编写内存分配函数，你要记住，在缺省情况下，C++ 在全局范围提供如下形式的 operator new：

```
void* operator new(std::size_t) throw(std::bad_alloc);      // normal new

void* operator new(std::size_t, void*) throw();             // placement new

void* operator new(std::size_t,                             // nothrow new —
                   const std::nothrow_t&) throw();          // see [Item 49](http://blog.
```

如果你在一个 class 中声明了任何 operator news，都将覆盖所有这些标准形式。除非你有意防止 class 的客户使用这些形式，否则，除了你创建的任何自定义 new 形式以外，还要确保它们都可以使用。当然，还要确保为每一个你使其可用的 operator new 提供相应的 operator delete。如果你要这些函数具有通常的行为，只需要让你的 class-specific（类专用）版本去调用 global（全局）版本即可。

达到这种效果的一个简单方法是创建一个包含 new 和 delete 的全部常规形式的 base class（基类）：

```
class StandardNewDeleteForms {
public:
  **// normal new/delete**
  static void* operator new(std::size_t size) throw(std::bad_alloc)
  { return ::operator new(size); }
  static void operator delete(void *pMemory) throw()
  { ::operator delete(pMemory); }

  **// placement new/delete**
  static void* operator new(std::size_t size, void *ptr) throw()
  { return ::operator new(size, ptr); }
  static void operator delete(void *pMemory, void *ptr) throw()
  { return ::operator delete(pMemory, ptr); }

  **// nothrow new/delete**
  static void* operator new(std::size_t size, const std::nothrow_t& nt) throw()
  { return ::operator new(size, nt); }
  static void operator delete(void *pMemory, const std::nothrow_t&) throw()
  { ::operator delete(pMemory); }
};
```

想要在标准形式之外增加自定义形式的客户就能够使用 inheritance（继承）和 using declarations（使用声明）（参见 Item 33）来得到标准形式：

```
class Widget: public StandardNewDeleteForms {           // inherit std forms
public:
   using StandardNewDeleteForms::operator new;          // make those
   using StandardNewDeleteForms::operator delete;       // forms visible

   static void* operator new(std::size_t size,          // add a custom
                             std::ostream& logStream)   // placement new
     throw(std::bad_alloc);

   static void operator delete(void *pMemory,           // add the corres-
                               std::ostream& logStream) // ponding place-
    throw();                                            // ment delete
  ...
};
```

Things to Remember

- 在编写一个 operator new 的 placement 版本时，确保同时编写 operator delete 的相应的 placement 版本。否则，你的程序可能会发生微妙的，断续的 memory leaks（内存泄漏）。

- 当你声明 new 和 delete 的 placement 版本时，确保不会无意中覆盖这些函数的常规版本。

附录 A. 超越 Effective C++

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Effective C++ 覆盖了我认为对当前的 C++ 程序员最重要的通用指导方针，但是如果你有兴趣在更多的方面提升你的效力，我推荐你去研读我的其他 C++ 书籍，More Effective C++ 和 Effective STL。

More Effective C++ 覆盖其它的编程指导方针，并包括像效率和带有异常编程这样的话题的广泛讨论。它也记述了像 smart pointers（智能指针），reference counting（引用计数）和 proxy objects（代理对象）这样的重要的 C++ 编程技术。

Effective STL 像 Effective C++ 一样是一本面向指导方针的书，但是它专注于标准模板库的有效使用。

下面是这两本书的目录摘要。

Contents of More Effective C++

Item 11: Prevent exceptions from leaving destructors

Item 12: Understand how throwing an exception differs from passing a parameter or calling a virtual function

Item 13: Catch exceptions by reference

Item 14: Use exception specifications judiciously

Item 15: Understand the costs of exception handling

Efficiency

Item 16: Remember the 80-20 rule

Item 17: Consider using lazy evaluation

Item 18: Amortize the cost of expected computations

Item 19: Understand the origin of temporary objects

Item 20: Facilitate the return value optimization

Item 21: Overload to avoid implicit type conversions

Item 22: Consider using op= instead of stand-alone op

Item 23: Consider alternative libraries

Item 24: Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI

Techniques

Item 25: Virtualizing constructors and non-member functions

Item 26: Limiting the number of objects of a class

Item 27: Requiring or prohibiting heap-based objects

Item 28: Smart pointers

Item 29: Reference counting

Item 30: Proxy classes

Item 31: Making functions virtual with respect to more than one object

Miscellany

Item 32: Program in the future tense

Contents of Effective STL

Item 20: Specify comparison types for associative containers of pointers.

Item 21: Always have comparison functions return false for equal values.

Item 22: Avoid in-place key modification in set and multiset.

Item 23: Consider replacing associative containers with sorted vectors.

Item 24: Choose carefully between map::operator[] and map::insert when efficiency is important.

Item 25: Familiarize yourself with the nonstandard hashed containers.

Chapter 4: Iterators

Item 26: Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator.

Item 27: Use distance and advance to convert a container's const_iterators to iterators.

Item 28: Understand how to use a reverse_iterator's base iterator.

Item 29: Consider istreambuf_iterators for character-by-character input.

Chapter 5: Algorithms

Item 30: Make sure destination ranges are big enough.

Item 31: Know your sorting options.

Item 32: Follow remove-like algorithms by erase if you really want to remove something.

Item 33: Be wary of remove-like algorithms on containers of pointers.

Item 34: Note which algorithms expect sorted ranges.

Item 35: Implement simple case-insensitive string comparisons via mismatch or lexicographical_compare.

Item 36: Understand the proper implementation of copy_if.

Item 37: Use accumulate or for_each to summarize ranges.

Chapter 6: Functors, Functor Classes, Functions, etc.

Item 38: Design functor classes for pass-by-value.

Item 39: Make predicates pure functions.

Item 40: Make functor classes adaptable.

Item 41: Understand the reasons for ptr_fun, mem_fun, and mem_fun_ref.

Item 42: Make sure less<T> means operator<.

Chapter 7: Programming with the STL

Item 43: Prefer algorithm calls to hand-written loops.

Item 44: Prefer member functions to algorithms with the same names.

Item 45: Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range.

Item 46: Consider function objects instead of functions as algorithm parameters.

Item 47: Avoid producing write-only code.

Item 48: Always #include the proper headers.

Item 49: Learn to decipher STL-related compiler diagnostics.

Item 50: Familiarize yourself with STL-related web sites.

附录 B. 第二和第三版之间的 Item 映射

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Effective C++ 的第三版在很多方面与第二版不同，其中最引人注目的是它包含很多新的信息。然而，第二版的大部分内容依然保留在第三版中，虽然经常会改变形式和位置。在后面几页的表格中，我展示了第二版 Items 中的信息在第三版的哪里可以找到，反之亦然。

这个表展示了一个信息的映射，而不是文本的。例如，第二版的 Item 39 中的思想（“避免在继承体系中做向下转型 (cast down) 动作”）（此标题借用侯捷先生的第二版译文——译者注）现在可以在当前版本的 Item 27（“最少化 casting（强制转型）”）中找到，即使第三版这个 Item 的文本和例子完全是新的。一个更极端的例子在于第二版的 Item 18（“努力让接口完满 (complete) 且最小化”）（此标题借用侯捷先生的第二版译文——译者注）。那个 Item 的主要结论之一是：不需要对 non-public（非公有）构件进行特殊访问的 prospective member functions（候选成员函数）一般应该成为 non-members（非成员）。在第三版中，通过不同的（更强的）论证达到相同的结果，所以第二版中的 Item 18 映射到第三版中的 Item 23（“用 non-member non-friend functions（非成员非友元函数）取代 member functions（成员函数）”），即使这两个 Item 之间仅有的共同之处是它们的结论。

Second Edition to Third Edition

2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.
1	2	18	23	35	32
2	-	19	24	36	34
3	-	20	22	37	36
4	-	21	3	38	37
5	16	22	20	39	27
6	13	23	21	40	38
7	49	24	-	41	41
8	51	25	-	42	39
9	52	26	-	43	44, 40
10	50	27	6	44	-
11	14	28	-	45	5
12	4	29	28	46	18
13	4	30	28	47	4
14	7	31	21	48	53
15	10	32	26	49	54
16	12	33	30	50	-
17	11	34	31		

Third Edition to Second Edition

3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.	3rd Ed.	2nd Ed.
1	-	20	22	39	42
2	1	21	23, 31	40	43
3	21	22	20	41	41
4	12, 13, 47	23	18	42	-
5	45	24	19	43	-
6	27	25	-	44	42
7	14	26	32	45	-
8	-	27	39	46	-
9	-	28	29, 30	47	-
10	15	29	-	48	-
11	17	30	33	49	7
12	16	31	34	50	10
13	6	32	35	51	8
14	11	33	9	52	9
15	-	34	36	53	48
16	5	35	-	54	49
17	-	36	37	55	-
18	46	37	38		
19	pp. 77-79	38	40		